RADC-TR-72-39
Final Technical Report
February 1972

INTERACTIVE DISPLAY LANGUAGE (IDL)

Informatics, Incorporated

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author) Informatics, Incorporated 6000 Executive Boulevard Rockville, Maryland 20852	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP N/A

3 REPORT TITLE

INTERACTIVE DISPLAY LANGUAGE (IDL)

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)

Final Report 1 Nov 1970 - 30 Nov 1971

5 AUTHOR(S) (First name, middle initial, last name)

Claude T. Creswell
Aldo L. DiPasqua
Fred F. Mitchell

6 REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
February 1972	226	0

8a. CONTRACT OR GRANT NO	9a. ORIGINATOR'S REPORT NUMBER(S)
F30602-71-C-0092 Job Order No. 69DB0000	TR-71-1232-1 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) RADC-TR-72-39

10 DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11 SUPPLEMENTARY NOTES	12 SPONSORING MILITARY ACTIVITY
None	Rome Air Development Center (IRDA) Griffiss Air Force Base, New York 13440

13 ABSTRACT

This report describes the implementation of an interactive programming language that allows a user to specify and compile programs in an interactive mode using an on-line console.

D.) FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Interactive Programming						
Pseudo Computer						
Interpretive Language						
CDC-1700						
BR-90						

INTERACTIVE DISPLAY LANGUAGE (IDL)

Claude T. Creswell
Aldo L. DiPasqua
Fred F. Mitchell

Informatics, Incorporated

FOREWORD

This report was prepared by Claude T. Creswell, Aldo L. DiPasqua, and Fred F. Mitchell of the Rome, New York, office of Informatics, Incorporated, 6000 Executive Boulevard, Rockville, Maryland. Its purpose is to document work accomplished under contract F30602-71-C-0092, Job Order Number 69DB0000, for Rome Air Development Center, Griffiss Air Force Base, New York. Submission of this report is in conformance with the requirements of subject contract Exhibit Line Item A003 and Section 5.2 of the Statement of Work. The work described herein was performed between 1 November 1970 and 3 November 1971 under the guidance and administration of RADC Project Engineer Garry W. Barringer (IRDA).

The objective of this effort was the implementation of the Bunker Ramo Interactive Display Language (IDL) written for the BR-133 on the CDC 1700 at RADC.

This report has been reviewed by the Information Office (OI) and is releasable to the National Technical Information Service (NTIS).

This technical report has been reviewed and is approved.

Approved: Garry W. Barringer
GARRY W. BARRINGER
Project Engineer

Approved: Franz H. Dettmer
FRANZ H. DETTMER
Colonel, USAF
Chief, Intel and Recon Division

FOR THE COMMANDER: Fred I. Diamond
FRED I. DIAMOND
Acting Chief, Plans Office

ABSTRACT

This report describes the implementation of an interactive display language that allows a user to specify and compile programs in an interactive mode using an on-line console.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF APPENDICES

LIST OF FIGURES

LIST OF FIGURES (Con'td)

LIST OF FIGURES (Cont'd)

SECTION 1
INTRODUCTION AND BACKGROUND

Contract AF30602-71-C-0092 "Interactive Display Language" is based on interactive display techniques developed by the Bunker Ramo Corporation from 1963 thru 1970. The current effort is the transfer of these techniques from a BR-133 computer to a CDC-1700 computer at Rome Air Development Center.

The original concept for an interactive programming system was developed by Culler and Fried at TRW (later BR) in 1963. The original implementation was based on the RW-400 Polymorphic computer and a Display Analysis Console (DAC). A later version used a BR-85 display console and a CDC-1604 computer. The final or 1968 version was a BR-133/230 complex at Canoga Park, California interfaced via a telephone line data link with a BR-90 at RADC.

Although IDL has been in existence for almost seven years, it has changed so many times that it has been difficult to judge the real utility of its interactive techniques.

The purpose then of the current effort has been to implement the 1968 version as described in existing documentation on the RADC CDC-1700 computer. Another part of the effort has been an evaluation of IDL. The results of this evaluation have lead to the conclusions in Section 11. Section 2 - 10 describe the various aspects of the software that make up the CDC-1700 version of IDL.

SECTION 2

IDL SYSTEM SUMMARY

2.1 IDL ELEMENTS

IDL is an operating system which allows communication between a BR 90 console and a central computer. This operating system consists of a computer language and the psuedo computer on which this language is assembled and executed. Two software programs, the Interpreter and the Command Routines, are the psuedo computer. Together they perform the interpretive functions normally executed by a computer's internal hardware. The use of a psuedo computer allows a certain degree of machine independence in the IDL language itself.

The language of the user is based on a 12 bit code generated by the Bunker Ramo 90 display (BR 90). This code is referred to as a key signature. The signatures are generated when any one of the 30 keys on the variable function keyboard is depressed. There are 64 overlays for the keyboard. Keys and overlays in combination will produce a total of 1920 unique 12 bit signature codes. The key signatures associated with overlays 0-6 are used to define the Basic IDL Language functions. The remainder of the key signatures are reserved for assignment as storage labels or program names.

Figure 2-1 lists the subset of the original Bunker Ramo IDL functions which were implemented in this effort. It contains all the essential functions to provide a preliminary version of IDL.

2.2 SYSTEM STRUCTURE

To reduce complexity and increase the efficiency of the existing IDL, it was decided to base the design of IDL on table driven software modules. A table system solved the problem of how to group keys with like properties together. For example keys which function

Key Num	Overlay 000	Overlay 001	Overlay 002	Overlay 003	Overlay 004	Overlay 005	Overlay 006
01	OV+1	OV+1	OV+1	OV+1	OV+1	OV+1	OV+1
02	OV-1	OV-1	OV-1	OV-1	OV-1	OV-1	OV-1
03	SR	X9REG	unused	unused	BP1	DECFL	RDMTB
04	HFLA	X8REG	unused	unused	BP2	DEC	RDEOF
05	FLAH	X7REG	unused	LTR	BP3	OCT	RWNMT
06	ADDFL	ADDFX	unused	SETLI	BP4	START	TRANS
07	SUBFL	SUBFX	RDOP	ALARM	BP5	STOP	XCHNG
10	SINE	X6REG	WROP	ERRCV	BP6	SREG	WRMTB
11	ARCTAN	X5REG	DATIN	RBLTR	ABORT	DECPT	WREOF
12	LOGAR	X4REG	OATOV	XBLK	ZOPR	NEG	BKSP
13	MULFL	MUL	unused	unused	unused	DEFIN	FLACC
14	DIVFL	DIV	unused	unused	unused	END	FXACC
15	COSINE	X3REG	unused	unused	unused	9	unused
16	SQROOT	X2REG	unused	DMFR	LNPRT	8	DMB
17	ANTLOG	X1REG	unused	DMTO	unused	7	MTB

IDL Function Keys

Figure 2-1

Key Num	Overlay 000	Overlay 001	Overlay 002	Overlay 003	Overlay 004	Overlay 005	Overlay 006
20	JFLPOS	JFXPOS	unused	CLM	DISPL	SHLR8	CURX
21	UNDIG	JFXZRO	unused	unused	CALL	LF	CURY
22	JFLNEG	JFXNEG	unused	unused	unused	6	LGR
23	CHSFL	INCR	unused	RDMT	A2KOP	5	RPK
24	ABSFL	DECR	unused	WRMT	unused	4	BLKTR
25	CFLG	CMPGFX	unused	SRCHNH	PAUSE	ASIGN	unused
26	CAE	CMPEFX	unused	SRCHEQ	unused	UNDEF	unused
27	CFLL	CMPLFX	unused	SRCHNL	ASY2K	3	WRLP
30	SETRU	FX2CMP	unused	SRCHNQ	READA	2	unused
31	SETFL	FX1CMP	unused	unused	WRTA	1	unused
32	JFLOV	JTRU	unused	CPC	READB	EXECU	RDCD
33	JFXOV	JFLS	SHLL8	SM	WRTB	SUBST	RDCUR
34	A2KMT	JALL	unused	unused	unused	INFIN	unused
35	HFXA	XTRCT	STFX	SC	unused	0	RDDSP
36	FXAH	MT2K	LDFX	unused	unused	/	WRDSP

Figure 2-1 (Cont'd)

as system directives (ASSIGN, DEFINE, EXECUTE, etc) can be easily differentiated from operating commands (add, subtract, multiply, etc) and/or other types. Table driven software also allows the programs which handle key signatures to be generalized in nature. Specific differences in the functioning of a generalized routine are controlled by the table entries for the key being handled. Another aspect of table driven software is that modification to the basic IDL may be made with only a minor change to the existing tables.

To deal with the problem of machine independence IDL was made to operate in a interpretive mode. Code is only translated into machine functions at execution time. Therefore only the Command Routines are bound to the limitations of a given compiler. The basic design could be implemented on any machine. The only recoding necessary would be that required to conform to the new instruction set. However a completely new set of Command Routines might be necessary since manufactures do not always conform to the same I/O conventions.

Four system modules and a series of Command Routines are used in the implemented IDL. They are as follows:

- o IDL Executive
- o IDL Assembler
- o IDL Loader
- o IDL Interpreter
- o IDL Command Routines

2.2.1 IDL Executive

The Executive module controls all scheduling and user communication. Entry to all the other system modules is possible only through the Executive. The main function of the Executive is to control the mode of the system. There are 5 modes which are controlled by the Executive: IDLE, DEFINE, ASSIGN, ASSEMBLE and EXECUTE. The Executive is in the IDLE mode when there are no

signature keys to be processed. The user is responsible for setting all modes except ASSEMBLE. The user sets the proper mode by depressing the key at the BR 90 console which has been defined as a system directive function key (i.e. overlay 5 key 13 for Define). The Exit procedure from any mode except Execute automatically sets the mode to ASSEMBLE. Keys which set the system modes are part of a set of keys called system directives. After a system directive has been received and processed the next key signatures received are passed to the appropriate system module. For example, after a Define directive key is processed the next key will be sent to the Assembler. After the execute key the next key is checked with the key identification table to be sure it is a program key then it is passed to the Loader and the execution process begins.

There are other specialized system directives that the Executive looks for; their function is described in appendix A, and in the detailed Executive program description. One last function of the Executive which is not controlled by the user is error abort. When an error occurs a message defining that error is sent to the operator. Then the original tables are replaced in core. This allows the user to recompile the program after correction without undefining all the keys equated in the erroneous pass.

2.2.2 Assembler Module

The Assembler module receives all its inputs via the Executive. The Assembler performs three basic functions based on mode. In the DEFINE mode the Assembler defines the subject key as a program key. The key identification table will be updated with this information so that future reference to that key will define an entire IDL user program.

In the ASSIGN mode storage assignments are made.

Storage may be in the form of single word locations or blocks of any length. If desired, storage areas may be preset with data.

As assignments are made the key signatures are packed away into a storage packet. (Section 5).

In the ASSEMBLE mode each key is checked against the key identification table for type. The Assembler is basically concerned with two types of keys. They are commands and parameters. Parameters take the form of registers, blocks, and constants. A Command key is checked for its syntax (Section 3). This syntax is used to determine the legality of the parameters received. As Commands with their associated parameters are compiled, they are put into the Command packet. (Section 5.6).

When syntax errors are found an error message will be generated informing the user of his error. At this point he may re-enter the Command in error or abort the assembly process.

The assembly process is ended when the Assembler detects an "End" key following a completed Command. At this point the Assembler combines all the assembled Loader packets (Section 5.1). The disk Loader information is compiled and appended to the beginning of the storage packets. The newly assembled program is written on the disc. The key identification table is updated to reflect the changes. Control is once again passed back to the Executive module.

2.2.3 IDL Loader

The IDL Executive detects the execute key and begins the execute process by loading a user defined program. The program key follows the Executive key and its validity is checked by the Executive. The key signature is sent to the Loader which obtains the disk address of the program from the key identification table.

The Loader reads in from the disk the storage, constants, and jump packets first. The Loader equates the ass storage keys to an absolute core location. Constants and jump k table entries are updated to reflect their respective core locations. Next the user generated Command string is read into core. The Command string is the users IDL program. The entire program is checked for missing labels of undefined keys. Finally the starting location of the Command string is sent to the Executive which takes control.

2.2.4 IDL Interpreter

The Interpreter controls the execution of the Command string. The Executive passes to the Interpreter the starting location of the Command string. The Interpreter reads up the Command, checks its syntax, and forms the effective address. Each Command has an associated syntax type listed in the key description table (Section 4.1). This syntax tells the Interpreter how to build the effective address block upon which the Command will act. A more detailed description of the effective address block may be found in Section 5. The IDL Interpreter obtains the entry address of the indicated Command Routine from the key description table. Control is passed to the Command Routine by means of a return jump to that entry address. Upon return from the Command Routine the Interpreter reads up another Command and the process repeats.

There are several Command functions which the Interpreter handles itself. All program branching is done by the Interpreter which includes the checking of the conditions necessary for a jump. For example, if a branch on overflow is called for, the Interpreter will check the overflow indicator for a true or false condition. Break points are also handled by the Interpreter. All breakpoints cause the

program flow to halt. A pause encountered by the Interpreter will cause a program halt. The execution of the Command string will continue until the Interpreter decodes an "End" key (i.e. overlay 5 key 14). The "End" key signals the end of the user program and control is returned to the Executive module.

2.2.5 IDI Command Routines

Command Routines are single purpose subroutines which perform the function described by the individual keys. Originally key functions were equated to the instruction set of a Bunker Ramo computer. Because of the differences in computer instruction sets it is now necessary to accomplish the function by means of a subroutine. Some routines perform specific tasks which do not require parameters. Most routines perform their function upon user selected targets. Upon entry to the routine it must first determine the target or effective address. The Interpreter has passed to the routine a block of data called the effective address block. Using its own syntax (Section 3) the Command forms the address and proceeds to act upon it. When its assigned task is complete, control returns to the Interpreter. A full discussion of each Command Routine can be found in Section 10.

SECTION 3

IDL SYNTAX

3.1 GENERAL

IDL Commands may have a variety of parameters. In order to ascertain the validity of any given parameter a table of legal syntaxes was needed. A study of each of the Command Routines was undertaken. The results indicated that the Commands could be classified in seven categories. Tables were built to cover the unique parameter types which made up each of the seven categories. These table entries are referred to throughout this report as the syntax type.

3.2 ELEMENTS OF SYNTAX

Five basic parameter types make up the allowable syntax. These are described below. In all cases the parameters are key signatures. These key signatures are used to define core locations or constants. The syntax is used to determine the function of these being keys in a given situation. There are several keys which represent fixed locations of core. This is by definition of the basic IDL. These keys are outlined and defined in Section 4. Keys defined in this way refer to the common data area which make up the psuedo computers working storage. This block of core is part of the basic IDL program. Unlike user defined storage, IDL working storage always occupies the same area of core and is not processed by the Loader. Keys may be assigned by the user to represent locations of the memory. This type of key may only be used with the program that defined it. The assignment of an absolute core location of a user defined key will be made by the Loader.

When constants are to be parameters they will take the form of an index into a constant packet (Section 5.4).

The constants used by the user program will have been defined using the special constant generating keys on overlay 5. Once the user has defined a constant it is placed in the constant packet and a pointer to it placed in the Command string.

Since most parameters are nothing more than core locations or constants, how they are used becomes very important. Bunker Ramo defines the five basic uses of parameter as follows:

1. FXOP - Fixed Point Operand. This single 16 bit data word is used as storage location. It can be an IDL defined psuedo register or some user defined storage. The FXOP may be part of another block but it is always a specific location within that block.
2. BLOCK - The parameter represents the first word address of a group of sequential 16 bit words. Blocks occur in basic IDL storage and may be created by the user. They are used for storage.
3. INDEX - A single 16 bit location. This may be the address of a IDL psuedo register, a constant or user defined key storage location. The index is used to specify a location within a block of data.
4. OPN - The address of a single location which contains the count which is to be used in a block operation. The OPN may be any one of the IDL defined psuedo registers, a constant, or user assigned storage.
5. FLOP - Floating Point Operand. The FLOP is a set of two sequential 16 bit words. The FLOP is used to contain Floating point data The two words

may be part of an IDL defined block or a user assigned block of storage. FLOP must always specifically represent the first word address of the set.

All IDL keys have an identification code. This 4 bit code classifies the key as one of 8 operational types. A 4 bit hexidecimal number is appended to the key signature entry in the key identification table (Section 4.1.1). This code is the element that is checked for syntactical validity.

Identification codes are as follows:

0 Unused key

1 IDL Directive

2 IDL Command

3 IDL Jump instruction

4 IDL one word register

6 IDL Program Name

8 IDL Storage Block

F A constant

3.3 DETAILED SYNTACTICAL STRUCTURE

There are seven legal syn ax types in IDL. All IDL Commands, key type 2, are followed by parameters. The parameters must agree with the syntax type associated with that key signature. The syntax type of each IDL Command is located in the appropriate entry of the key description table (Section 4.1.2).

The seven syntax types are as follows:

- Type 0 syntax

  No parameter allowed

- Type 1 syntax

  /BLOCK /INDEX/OPN/BLOCK/INDEX/OPN/

- o Type 2 syntax

  /FXOP/ FXOP/
- o Type 3 syntax

  /BLOCK/Index/Opn/
- o Type 4 syntax

  FXOP or FLOP
- o Type 5 syntax

  Program Key Only
- o Type 6 syntax

  Another IDL Command only

Using the data described in Section 3.2, Figures 3.1 through 3.6 show a breakdown of these syntax types into the allowable options of each. The options take the form of a sequential series of key types. Syntax has been described in general terms. Each of these general terms can have several specific key series representations. For example, a FXOP can be an IDL defined psuedo register (key type 4). However, it can also be a block (key type 8) which would require an index into it. The index itself can be a register (key type 4) or it could be a constant (key type F). Therefore this FXOP could be represented by 3 different key series (type 4, or type 8 followed by type 4, or type 8 followed by type F).

Sections 3.3.1 to 3.3.7 describe the syntax which is represented in the syntax option tables which make up Figures 3.1 through 3.6. Figures 3.1 through 3.6 are graphic representations of the actual syntax table as it appears in core. The first location of each table contains the number of options available for that syntax type. Options are read from left to right or most significant to least. The options are left justified with trailing zeros. Each option type is represented as two 16 bit words. These words are divided into 8

4 bit fields. Each field contains an allowable key type. The first zero terminates the option series.

3.3.1 Syntax Type 0

This is a null type which applies to all IDL Commands which have no arguments or parameter.

3.3.2 Syntax Type 1

Fig 3-1 describes the 36 options available to this syntax. These options define all the legal combinations of parameters for the Commands. Its general format is Block, Index, Opn, Block, Index, Opn.

3.3.3 Syntax Type 2

Fig 3-2 describes the 9 legal options for this syntax type. Its general format is FXOP, FXOP, where FXOP is either Block, Index, Opn or a fixed point register (key type 4).

3.3.4 Syntax Type 3

Fig 3-3 describes the 6 legal options for syntax type 3. The general format is Block, Index, Opn.

3.3.5 Syntax Type 4

Fig 3-4 depicts this syntax. There are 4 legal options and the general form is FXOP or FLOP where either can be Block, Index, or an IDL register.

3.3.6 Syntax Type 5

Fig 3-5 shows this syntax. There is only 1 legal syntax and it must be key type 6.

3.3.7 Syntax Type 6

Fig 3-6 describes this syntax. Like type 5 it has only one legal format which is key type 2.

	entries = 36			
1	8	4	4	8
	4	4	0	
2	8	4	4	8
	4	F	0	
3	8	4	4	8
	4	0		
4	8	4	4	8
	F	4	0	
5	8	4	4	8
	F	F	0	
6	8	4	4	8
	F	0		
7	8	4	F	8
	4	4	0	
8	8	4	F	8
	4	F	0	
9	8	4	F	8
	4	0		
10	8	4	F	8
	F	4	0	
11	8	4	F	8
	F	F	0	
12	8	4	F	8
	F	0		
13	8	F	4	8
	4	4	0	
14	8	F	4	8
	4	F	0	
15	8	F	4	8
	4	0		
16	8	F	4	8
	F	4	0	
17	8	F	4	8
	F	F	0	
18	8	F	4	8
	F	0		
19	8	4	8	4
	4	0		
20	8	4	8	4
	F	0		
21	8	4	8	4
	0			
22	8	4	8	F
	4	0		
23	8	4	8	F
	F	0		
24	8	4	8	F
	0			
25	8	F	F	8
	4	4	0	
26	8	F	F	8
	4	F	0	
27	8	F	F	8
	4	0		
28	8	F	F	8
	F	4	0	
29	8	F	F	8
	F	F	0	
30	8	F	F	8
	F	0		
31	8	F	8	4
	4	0		
32	8	F	8	4
	F	0		
33	8	F	8	4
	0			
34	8	F	8	F
	4	0		
35	8	F	8	F
	F	0		
36	8	F	8	F
	0			

IDL Syntax Type 1 /BLOCK/INDEX/OPN/BLOCK/INDEX/OPN/

Figure 3-1

	entries = 9			
1	4	4	0	
	0			
2	4	8	4	0
	0			
3	4	8	F	0
	0			
4	8	4	4	0
	0			
5	8	4	8	4
	0			
6	8	4	8	F
	0			
7	8	F	4	0
	0			
8	8	F	8	4
	0			
9	8	F	8	F
	0			

IDL Syntax Type 2 /FXOP/FXOP,

Figure 3-2

	entries = 6			
1	8	4	4	0
	0			
2	8	4	F	0
	0			
3	8	4	0	
	0			
4	8	F	4	0
	0			
5	8	F	F	0
	0			
6	8	F	0	
	0			

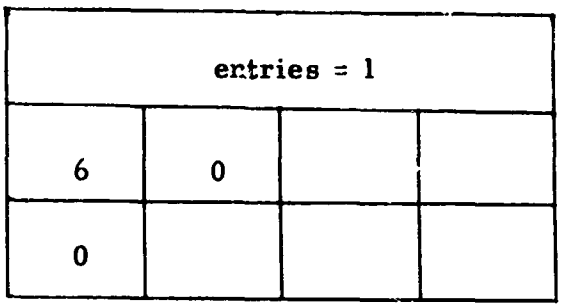
IDL Syntax Type 3 /BLOCK/INDEX/OPN/

Figure 3-3

entries = 4			
F	0	0	0
0	0	0	0
4	0	0	0
0	0	0	
8	4	0	
0	0	0	0
8	F	0	0
0	0	0	0

IDL Syntax Type 4 /Operand/

Figure 3-4



IDL Syntax Type 5 /Program/

Figure 3-5

entries = 1			
2	0		
0			

IDL Syntax Type 6 /Command/

Figure 3-6

SECTION 4

SYSTEM TABLES AND WORKING STORAGE

4.1 SYSTEM TABLES

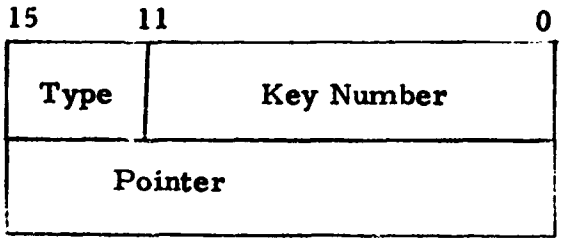
This section presents the design of the two tables used to control all phases of the system operation. They are the key identification table and the key description table. All software modules make repeated references to these tables. Modification of these tables is all that is necessary to add or delete functions of IDL.

4.1.1 Key Identification Table

Figure 4-1 presents the format of the key identification table entries. The table consists of 1920 two word entries, one for each of the 30 keys on all 64 overlays. The first word of the entry contains two fields. The least significant 12 bits contain the key signature. These 12 bits comprise a five bit key number and a seven bit overlay number. The remaining 4 bits contain a hexidecimal code to indicate the key type.

The key types are defined as follows:

- o Type 0 - Unused Key - All keys not used by either IDL itself or any application program.
- o Type 1 - IDL Directive - A system directive not allowed in a user program.
- o Type 2 - IDL Command - All IDL Command other than jumps.
- o Type 3 - IDL Jumps - All IDL Jump Commands and label follows.
- o Type 4 - IDL Registers - All IDL Single word IDL storage.



1920 Entries
Two Words Per Entry

Key Type	Definition	Pointer Contents
0	Unused Key	Blank
1	IDL Directive	Blank
2	IDL Command	Key Description Location
3	IDL Jump	Blank
4	IDL Register	A Core Location
6	IDL Program	Disc Location of Program
8	IDL Storage Block	A Core Location
F	IDL Constant	Key Description Location

IDL Key Identification Table

Figure 4-1

- o Type 5 - Not Used.
- o Type 6 - IDL Program - Any Key Defined by the user as a program.
- o Type 7 - Not Used.
- o Type 8 - IDL Storage Block - All IDL storage assigned as two or more words.
- o Type 9 thru E - Not Used.
- o Type F - IDL Constants - All IDL Constants or Constant generating functions.

The second entry of the key identification table is based on the function of the key type. In most cases the second word contains a pointer. The entry is blank for unused keys. This table entry will be filled by the assembler later when the key is defined. IDL Directives (key type 1) and jump commands (key type 3) do not need a pointer since they have a specific function. In their case the second entry is also blank. Commands and constants use the pointer to refer to another entry in the key description table (Section 4.1.2), which further describes their function. The pointer for both IDL Registers (key type 4) and blocks (key type 8) contain the absolute memory location of that register or blocks in core. In the case of basic IDL, defined registers and blocks, this address is set at system initialization time. For user defined registers and blocks the pointer will be filled in by the IDL loader. The pointer entry for program keys (key type 6) is the address of a disk sector which contains the user program. This pointer is built and stored by the assembler when a program is completed.

4.1.2 Key Description Table

The key description table contains supplemental information which is necessary to process IDL commands (key type 2).

This information is contained in a series of two word entries. The first word of each entry contains a pointer. The pointer identifies the entrance address of the associated command routine. This pointer is the only method of locating the command routines. The second word contains the applicable syntax type. This entry along with the syntax table (Section 3.3) will be used to check validity of the command parameters.

4.2 WORKING STORAGE

IDL working storage is logically broken into two categories. One category of storage is accessible only by the IDL operating system, while the other category contains storage which is available to the user programs.

The first category contains the locations within core which are used to compile a user program. The user cannot get to these areas of core. Some of these buffers exist only at compile time and are overlayed with the user program assignments when execution is called for. The core locations which make up this class of working storage are listed in Figure 4-2 with a discussion of each element following in Section 4.2.1.

Registers and buffers available to the user are part of the basic IDL. These registers and buffer areas all function as part of the pseudo computer. They replace the hardware registers one would normally find in a computer. A listing of this category is found in Figure 4-3 with a description following in Section 4.2.2.

Name	Length	Description
KEY	1	Signature of Key being processed
MODE	1	System Mode
CONLOC	1	Location of active constant packet
JMPLOC	1	Location of active Jump packet
SRTLOC	1	Starting location of user program
EFADBK	7	Effective Address Block
LODATA	7	Loader Data
DEFSTO	2000	Defined Storage Packet
COMPAC	9193	Block for user program
TRAVEC	200	Transfer Vector Packet
CONSTO	1000	Constant Storage Packet
PARAM	7	Parameter Block
JTAB	100	Jump Table
NEWKID	200	New Key ID Block
DSKBUF	200	Disk Buffer

SYSTEM REGISTERS & BUFFERS
Figure 4-2

4.2.1 IDL System Registers and Blocks

The following are descriptions of the elements found in Figure 4-2.

- o KEY — KEY NAME
  This register holds the signature of the key being processed.
- o MODE — SYSTEM MODE
  This is the mode of the system (assign, define, or assemble) as set by IDL Directive Keys (Key type 1).
- o CONLOC — CONSTANT LOCATION
  This register contains the absolute core address of the first constant packet.
- o JMPLOC — JUMP LOCATION
  This register contains the absolute core address of the first Jump packet.
- o SRTLOC — STARTING LOCATION
  This register will contain the starting address of the user defined program to be executed.
- o EFADBK — EFFECTIVE ADDRESS BLOCK
  This block is used to assemble the effective address of the data to be operated upon by the command routines. The interpreter puts the command parameters into this block then jumps to the command routine.

- LODATA — LOADER DATA
  This Block of core will hold the Loader data (Section 5.2).
- DEFSTO — DEFINED STORAGE
  This is where storage assignment information will be stored. It will also contain the storage packet (Section 5.3).
- CONSTO — CONSTANT STORAGE PACKET
  This block will hold constants as they are generated by the user. Its contents will become the constant packet (Section 5.4).
- TRAVEC — TRANSFER VECTOR PACKET
  This is the transfer vector packet storage area (Section 5.5).
- COMPAC — COMMAND PACKET
  This block is where the user program is assembled. When the assembly is finished that part which has been used will become the command packet (Section 5.6).
- PARAM — PARAMETER BLOCK
  This block holds the command parameters that are sent to the assembler until the applicable syntax is complete.
- JTAB — JUMP TABLE
  This block holds the labels which are the targets of user jumps. This table is checked against the final user program for missing JMP tables.

Name	Length	Description
FLACC	2 words	Floating Point Accumulator
FXACC	1 word	Fixed Point Accumulator
FXOVFL	1 word	Fixed Point Overflow Indicator
FLOVFL	1 word	Floating Point Overflow Indicator
X1REG	1 word	9 Fixed Point Index Registers
thru		
X9REG	1 word	
COMPAR	1 word	Comparator
SPLREG	1 word	Special Register
DMFR	1 word	Last Word Address of DMB
DMTO	1 word	Next to Last Word Address DMB
CURX	1 word	Cursor X Position
CURY	1 word	Cursor Y Position
LGR	1 word	Light Gun Register
LPPRN	1 word	Line Printer Register
A2KOP	2048 words	2K Operand
MTB	80 words	Magnetic Tape Buffer
DMB	80 words	Display Module Buffer
LTR	4 words	Light Register Buffer
PROJ	2 words	Projector Control Table
SATBUF	15 Words	BR-90 Status Buffer

APPLICATION PROGRAM REGISTERS AND BUFFERS

Figure 4-3

4.2.2 Application Program Registers and Blocks

The following registers are addressable by any application program. All registers are assumed to be one word, 16 bit registers except FLACC. These are found in Figure 4-3.

- o FLACC — FLOATING POINT ACCUMULATOR
  This is a two word 32 bit register. The results of all floating arithmetic will be in FLACC. It may be tested for jumps.
- o FXACC — FIXED POINT ACCUMULATOR
  The results of all fixed point arithmetic will be held in FXACC. It may be tested for jumps.
- o FXOVFL — OVERFLOW INDICATOR FIXED POINT
  This register will be set to Non-zero by any fixed point arithmetic operation that results in an overflow condition.
- o FLOVFL — OVERFLOW INDICATOR FLOATING POINT
  This register will be set to Non-zero by any floating point arithmetic operation that results in an overflow condition.
- o X1REG - X9REG — FIXED POINT INDEX REGISTERS
  There are 9 index registers that are available to the user.
- o COMPAR — COMPARATOR
  This register will be set true or false to be used in conjunction with compare commands.

o	SPLREG	SPECIAL REGISTER This register will contain a key signature to be used at executive time with the substitute key (0533).
o	DMFR	LAST WORD ADDRESS OF THE DMB This register will contain the address of last word in the Display Module Buffer.
o	DMTO	NEXT TO LAST WORD IN DMB This register will contain the last word address, less one, of the Display Module Buffer.
o	CURX	CURSOR X POSITION The current X coordinate of the display cursor is copied into this register on each occurrence of the "Read Cursor" (key 0633) operation.
o	CURY	CURSOR Y POSITION The current Y coordinate of the display cursor is copied into this register on each occurrence of the "Read Cursor" (key 0633) operation.
o	LGR	LIGHT GUN REGISTER The display memory address of the currently "Light Gunned" element on the display is copied into this register on each occurrence of the "Read Cursor" (key 0633) operation. (If the BR-90 does not have an active light gunned element, then zero is copied into this register.

- LRPRN — LINE PRINTER REGISTER
  This register may be set to provide a listing of the user program. The executive sets this register only after Key 0416 is depressed.
- A2KOP — 2K OPERAND BUFFER
  This is a 2048 word operand. It is used similar to any other operand and has the further capability of direct communication to or from a display console (see keys 0430, 0431, 0432 and 0433) or magnetic tape record (see keys 0034 and 0136). It is also used as the data source for the "Assemble 2K" and "Alternate Assemble 2K" functions (see keys 0427 and 0331).
- MTB — MAGNETIC TAPE BUFFER
  This 80 word buffer is used for communication with all I/O devices except the display consoles.
- DMB — DISPLAY MODULE BUFFER
  This 80 word buffer is used to communicate with the display console memory. The last two cells determine the memory addresses to be used in the display console. The last cell (79th or 117 octal) contains the beginning display address effected and the next to last cell (78th or 116 octal) contains the ending display address affected. (Also see DMFR and DMTO , keys 0316 and 0317).

- **LTR** — LIGHT REGISTER BUFFER

  This is a four-word buffer that controls the program key (variable function key) lights. Left keyboard 1-17 (octal) are controlled by bits 1-15 of the first word, Lights 20-36 (octal) by bits 1-15 of the second word. Right keyboard lights 1-17 (octal) are controlled by bits 1-15 of the third word, lights 20-36 (octal) by bits 1-15 of the fourth word. Bit 1 of the words are least significant and correspond to lights 1 and 20 (octal). A 1 in a bit position means light on while a 0 indicates light off. The setting of this four word buffer does not affect the actual lights until a "set lights" operation (Key 0306) is performed. Furthermore, the "I/O Buffer Controller" must be informed to stop automatic control of keyboard lights to make this operation meaningful.

- **PROJ** — PROJECTOR CONTROL TABLE

  A two word table used for control and selection of a slide for the rear projector in the BR-90.

- **SATBUF** — BR-90 STATUS BUFFER

  This is a 16 word buffer which will contain the current status of the BR-90 console. Figure 4-4 describes this buffer.

Word	Code		General Description
Word #1	0000	=	Indicates V. F. K. Code
	0001	=	Indicates Light Gun Code
	0002	=	Not recognized
	0003	=	Indicates Illegal Command
Word #2	xxxx	=	The code of the variable function keyboard
Word #3	0000	=	Indicates Random Mode Display
	0001	=	Indicates Formatted text mode of display
Word #4	0000	=	Main or Page A on Display
	0001	=	Format or Page B on Display
Word #5	0000	=	Indicates Maximum Display is off
	0001	=	Maximum Display is on
Word #6	xxxx	=	Address of circle or symbol or coordinates terminal point of vector
Word #7	0000	=	Indicates a symbol
	0001	=	Indicates a vector
Word #8	0000	=	Indicates that the cursor is on
	0001	=	Indicates that the cursor is off
Word #9	xxxx	=	"X" Coordinate of the cursor
Word #10	xxxx	=	"Y" Coordinate of the cursor
Word #11	xxxx	=	End of Message
Word #12	xxxx	=	End of Message
Word #13	xxxx	=	Location of marker in the current page
	0000	=	Indicates there is no marker
Word #14	Bits	=	Slide and Magazine Status
	0-3	=	BCD units digit of slide number (0-9)
	4-7	=	BCD tens digit of slide number (0-9)
	8-11	=	BCD Magazine Number (0-4)

Figure 4-4

Word #15	Bits	Projector Display Status
	0	Lower
	1	Center Y Offset
	2	Upper
	3	Right
	4	Center X offset
	5	Left
	6	0 = Magnified; 1 = Full-slide
	7	0 = Lamp off; 1 = Lamp on
	Remaining	Unused

Figure 4-4 (Cont'd)

SECTION 5

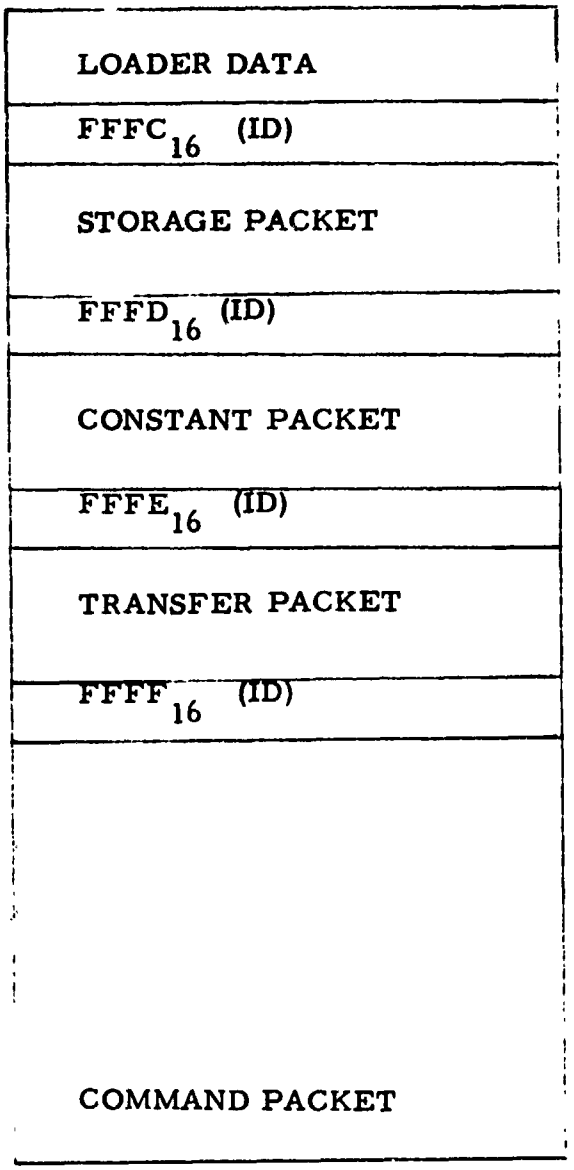
PROGRAM FORMATS

5.1 GENERAL

The IDL Loadable format is the basic internal format for all IDL user application programs. All user programs are converted to this format as they are assembled. The Loadable formatted program is comprised of five parts or packets. The five parts define Loader data, storage, constants, transfer labels and commands. The Assembler builds each packet as key signatures are received. The data contained in the Loadable format is in the form of key signatures, with the exception of constants. The constants are compiled by the Assembler into their respective hexidecimal value. When the application program has been successfully completed all packets are grouped together and written on the disk file. Figure 5-1 shows the general arrangement of the packets.

5.2 LOADER DATA

The first block of any assembled IDL program is the Loader data. This data is compiled by the Assembler. This data is used by the IDL Loader program to load the remainder of the user's program. The Loader data is made up of seven words. The first word of the block is the key signature which names the program. It is followed by the total number of words in the assembled program. The third word contains the number of words that make up the storage packet. The fourth word is the length of the constant packet followed by the length of the transfer packet in the fifth. The sixth and seventh words are used to describe the main body of the assembled IDL program. The sixth word is the length of the Command packet



LOADABLE FORMAT

Figure 5-1

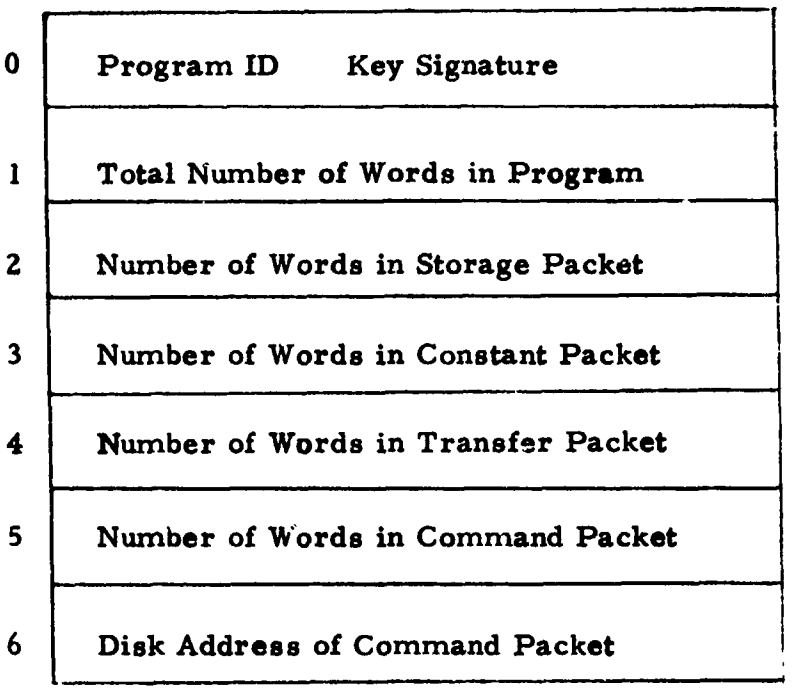
and the seventh word contains the Disk address of the Command packet. Figure 5-2 depicts the Loader data format.

5.3 STORAGE DATA

The first word of the storage packet contains the hexidecimal constant FFFC as an identifier. The second location contains the number of words within the packet. The remainder of the packet is filled with the information necessary for the Loader to assign the storage needed by the application program. This descriptive information takes the format of a key signature followed by a descriptor word. This descriptor will contain the amount of storage to be assigned to each key.

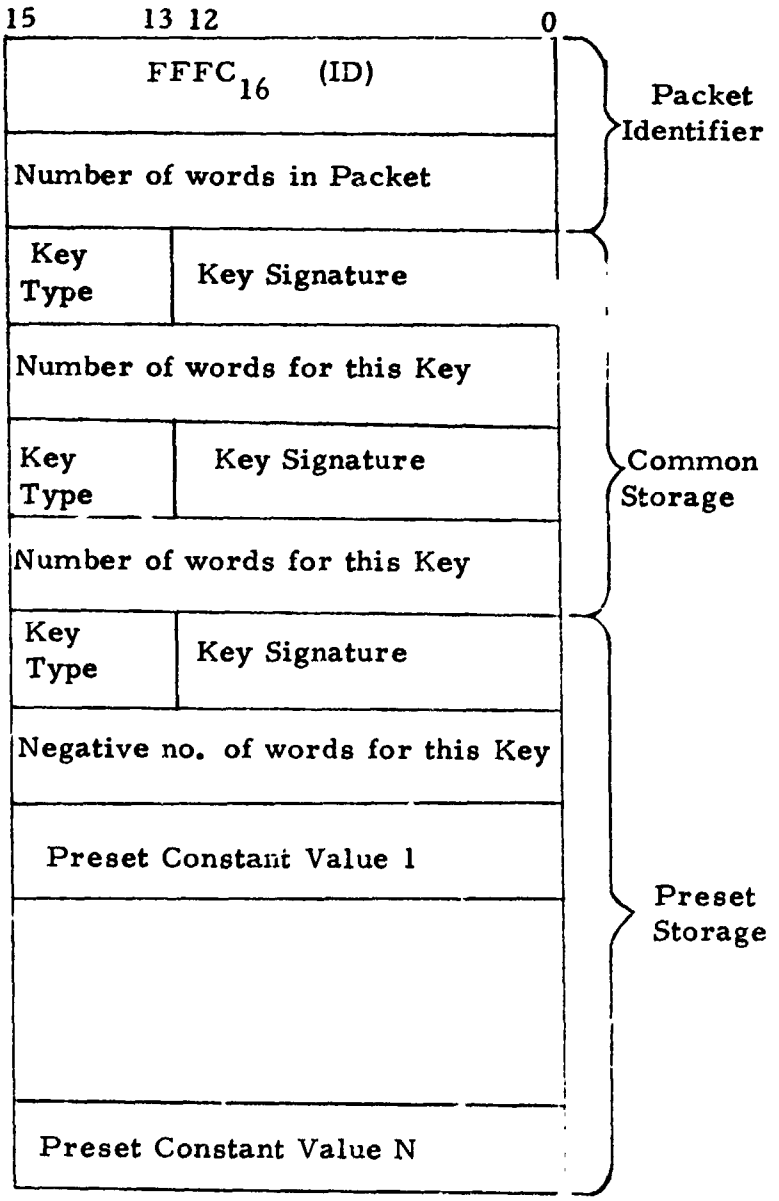
These are two types of storage which the user may assign. The first is referred to as blank or common storage. When common storage is indicated the descriptor word following the key signature will contain only the number of words to be assigned. The descriptor word will be positive. Common storage is assigned locations in core with no regard given to their contents.

The other type of storage which can be assigned is called "pre-set storage". In this case the descriptor will still contain the number of words to be assigned but it will be negative. That is, bit 15 of the descriptor word has been set to a 1. Preset storage assignment alters the contents of the core locations it is assigned to. The data which is to be placed in the assigned core follows the descriptor word in the storage packet. It should be noted here, if more core locations are assigned than there are data words, the remaining area is set equal to the last preset data word. Figure 5-3 represents the storage packet.



Loader Data Format

Figure 5-2



Storage Packet Format

Figure 5-3

5.4 CONSTANTS

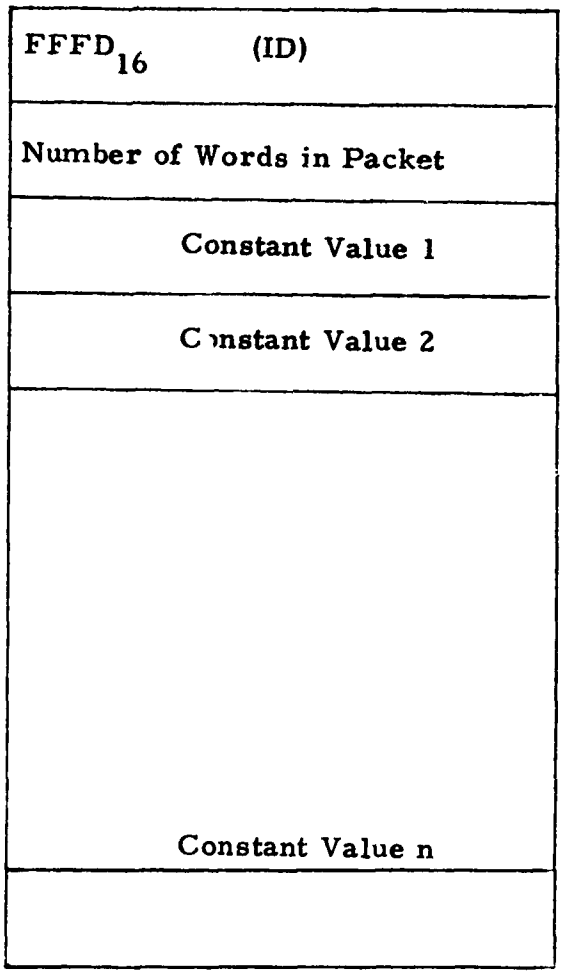
The constant values generated by the user during Assembly will be stored in a constant packet. An index is used to reference the constant values stored in the packet. When a Command uses a constant as a parameter the parameter is replaced by an index. The constant index is recognized by the hexidecimal number F in its most significant 4 bits. The lower 12 bits contain a number which identifies the constants location within the packet. For example, to reference the sixth word in the constant packet the Command parameter would be F006. All constants generated are checked against the constant packets. If a duplication occurs the index is set to the original entry.

Figure 5-4 represents the format of the constant packet. The first word is the hexidecimal number $FFFD_{16}$, the Identifier. The second word contains the number of words in the packet. The remainder of the packet is filled with constant values set by the user.

5.5 TRANSFER VECTOR FORMAT

The user may have jumps or branches within his application program. He can accomplish this during Assembly by using the "Label Follows" key (appendix A). This Command labels a key signature as the target or destination of the jump. This destination is stored in the transfer packet as a two word entry. The first word of entry will be the key signature defined as the target. The second word will be left blank.

Whenever an application program containing branching is loaded the IDL Loader routine will fill the second word for each target entry. At load time, a search of the Command string obtains the absolute core address of the label. This absolute address is stored



Constant Packet Format

Figure 5-4

in the transfer vector packet. Branching is done by the Interpreter. It searches the transfer packet for the proper label, reads up the address and performs its jump.

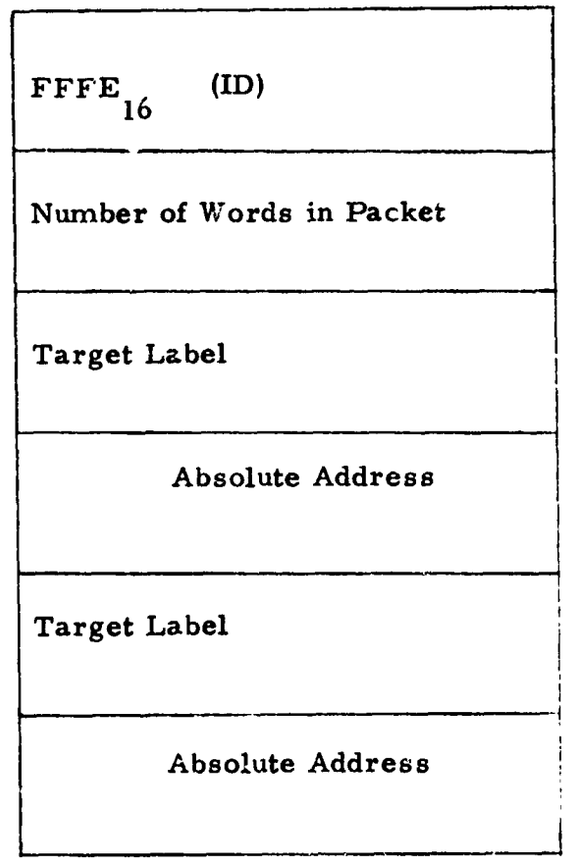
Figure 5-5 describes the format of the transfer vector packet. The first word contains the hexidecimal number FFFE as an Identifier. The second word is the number of words in the packet. The remainder of the packet is filled with the two word label identifiers.

5.6 COMMAND PACKET

The Command packet contains the actual IDL application program. This is the packet which will be translated to produce the desired results for the user. The primary contents of this packet are the Commands (key type 2), jumps (key type 3) and their associated parameters.

There are two types of formats within the Command packet. They are the Command key format and a format used for all the others. Any key which is not a Command (key type 2) will be represented in a two word format. The first word will be the key type and signature and the second word will contain its parameter. For example, the key signature for a "Label Follows" along with its key type (type 3) would occupy the first word. The second word will be the key signature of that key which is to be the label.

The other type or Command format will be a variable length format. The variable length format is necessary because of the different types of syntax which are used by IDL. The basic makeup for all Commands however will be the same. That is, each Command will have at least the first two words alike. The first word in every case will be the key type and signature, the second will be the syntax

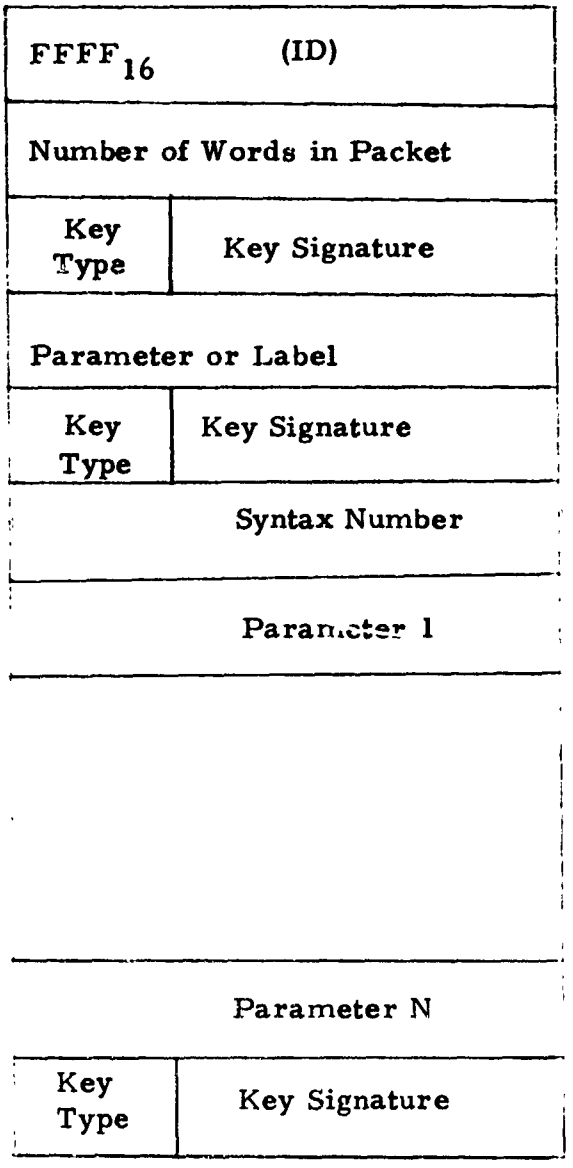


Transfer Vector Format

Figure 5-5

type. The parameters, if any, which have been assembled for that Command will follow these words. The number of words set aside for these parameters will be according to syntax length. Therefore, if a Command is a type 0 syntax (no parameters) it will appear within packet as a two word entry. Likewise, a type 2 syntax (six possible parameters) will be an 8 word entry.

Figure 5-6 shows the format of the Command packet. Like the other packets it contains a two word header. The hexidecimal constant FFFF identifies the packet and is followed by number of words in the packet.



Command Packet Format

Figure 5-6

SECTION 6

IDL EXECUTIVE PROGRAM

6.1 GENERAL

The Executive module controls all scheduling and user communication. Entry to all other modules is possible only through the Executive. The Executive idles while it waits of a key signature. Key signatures are read in when available and checked first for type.

The Executive processes only type 1 or system directive keys. If the key read in is not a type 1 the mode of the system is checked and the key sent to the proper module. When system directive keys are received the Executive performs the functions specified for that key. The Executive also checks for end of program key. This key is an IDL Command type 2, however a check is made to recognize it. Error recovery is also handled by the Executive. When an error occurs in one of the other modules a code is received by the Executive and it returns to IDLE mode to await operator intervention. The following discussion in 6.2 gives a detailed description of how each key is handled by the Executive. This description is followed by detailed function flow charts of the IDL Executive. (Figure 6-1)

6.2 DETAILED OPERATION

The Executive is in an idle loop until there is a key signature available to process. The first operation of the Executive upon receipt of a key signature is to obtain the key type from the Key Identification table. The key type is tested for an IDL Directive or type 1. There are five directives which are meaningful to the Executive. Many of the original directives were designed for hardware which is not available on the present system. For example, the paper tape reader and paper tape punch are not available, therefore, the directives which

dealt with them have been deleted. When one of the deleted functions is called for by the user the Executive simply No Ops the function. In this manner deleted functions may be added at a later date if the hardware becomes available.

The directives which are handled by the Executive are as follows:

1. DEFINE - overlay 5 key 13
2. ASSIGN - overlay 5 key 25
3. LNPRT - overlay 4 key 16
4. EXECUTE - overlay 5 key 32
5. START - overlay 5 key 7

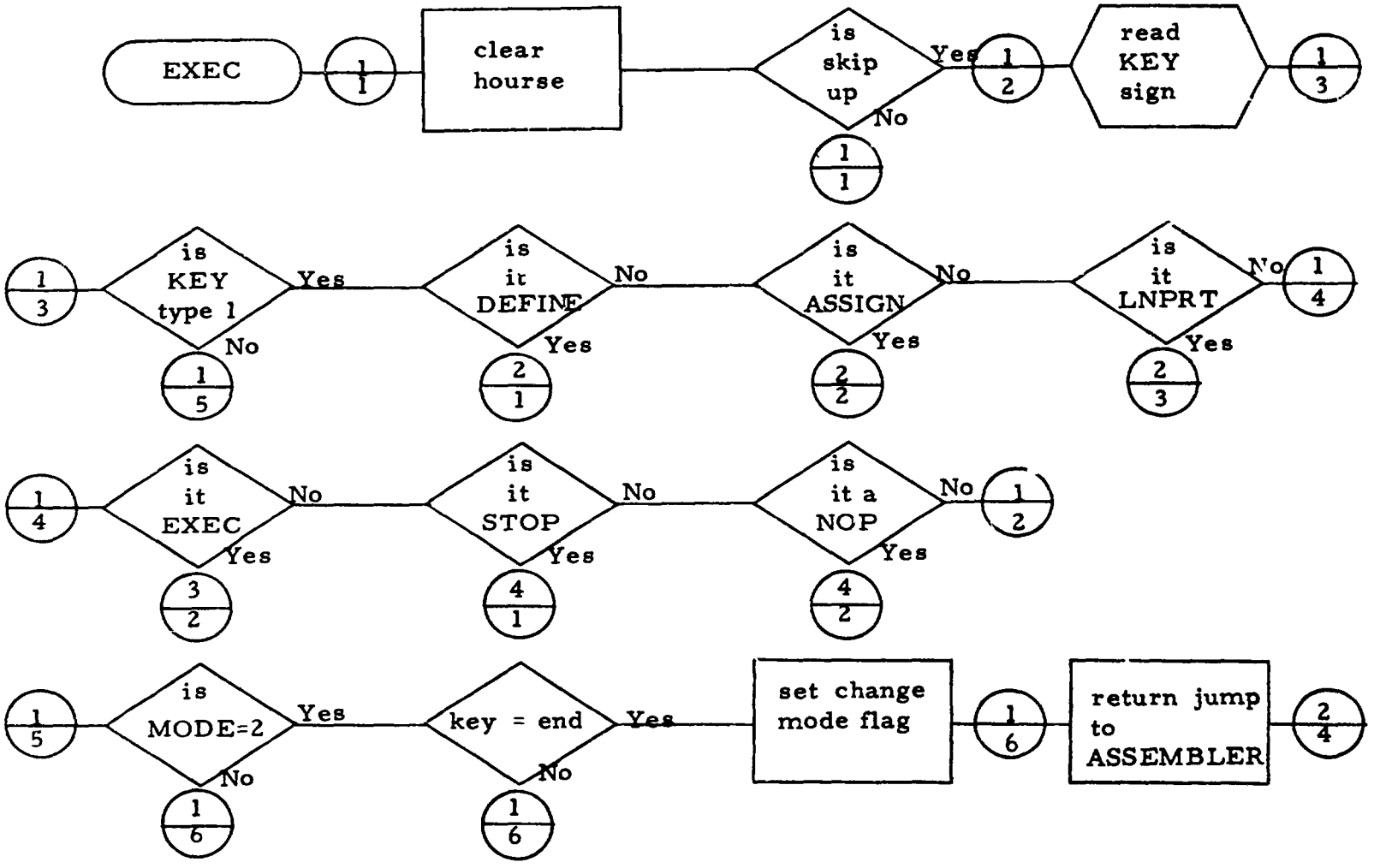
Two directive keys are responsible for the assembly of a new user program. They are the "Define" key which names the program and the "Assign" key which assigns storage. When the Executive receives a "Define" key it sets the mode to 3. This setting tells the Assembler that a new program is to be built. The "Assign" key causes the Executive to set the mode to 2 for storage assignment by the Assembler. The exit from the ASSIGN mode (mode 2) causes the Executive to go into ASSEMBLE mode (mode 1). This change is accomplished by checking each key signature, while in the ASSIGN mode, for the "End" key (see Assign-Appendix A). When an "End" key is detected the location "CHANGE" is set to one. The "End" key like all others is sent to the Assembler. When control returns from the Assembler and CHANGE is checked and found to be set, the Executive sets the mode to Assemble (mode 1). Once in Assemble mode the Executive does little more than pass key signatures to the Assembler.

The line printer directive, LNPR causes the Executive to set the print flag. After each key is processed, but before the next key signature is read in, the printflag is checked. If the flag is set the last key signature processed will be printed on the line printer.

This is the only form of a program listing available.

The "Stop" key will direct the Executive to update the permanent Key Identification table which is kept on the disk file. The core resident key ID table is updated by the Assembler as keys are assigned. If an error occurs the Assembly is aborted. Under the original concept, in order to use any of the same keys again, it was necesary to go through and undefine all the key defined in the erroneous pass. However, by not updating the disk copy of the Key ID table there will be no need to undefine the keys. This is so since the Executive reads in the disk Key ID table for each new Assembly. The "Stop" key should only be used when the user is satisfied that a program works correctly and wants to keep it in a permanent file.

The "Execute" key, the last of the IDL directives, cause a series of events to happen. First the Executive reads up the next key. This must be the name of an application program which has previously been defined. This key will carry a type 6 in the Key Identification table. If the Executive finds a program key it will place its signature in location "KEY" and call the Loader. The Loader will load the program and if no errors occurs it will return control to the Executive with the "A" register clear. In the case of a Loader error the return will be made with the "A" register set to 1. A check is made for an error condition. An error will abort the execution and the program will return to IDLE. An errorless return from the Loader is followed by a jump to the Interpreter for the actual execution of the application program. Again, upon return from the Interpreter a check for error condition is made, if error free, the message "JOB COMPLETE" is typed on the teletypewriter.



IDL Executive
Figure 6-1

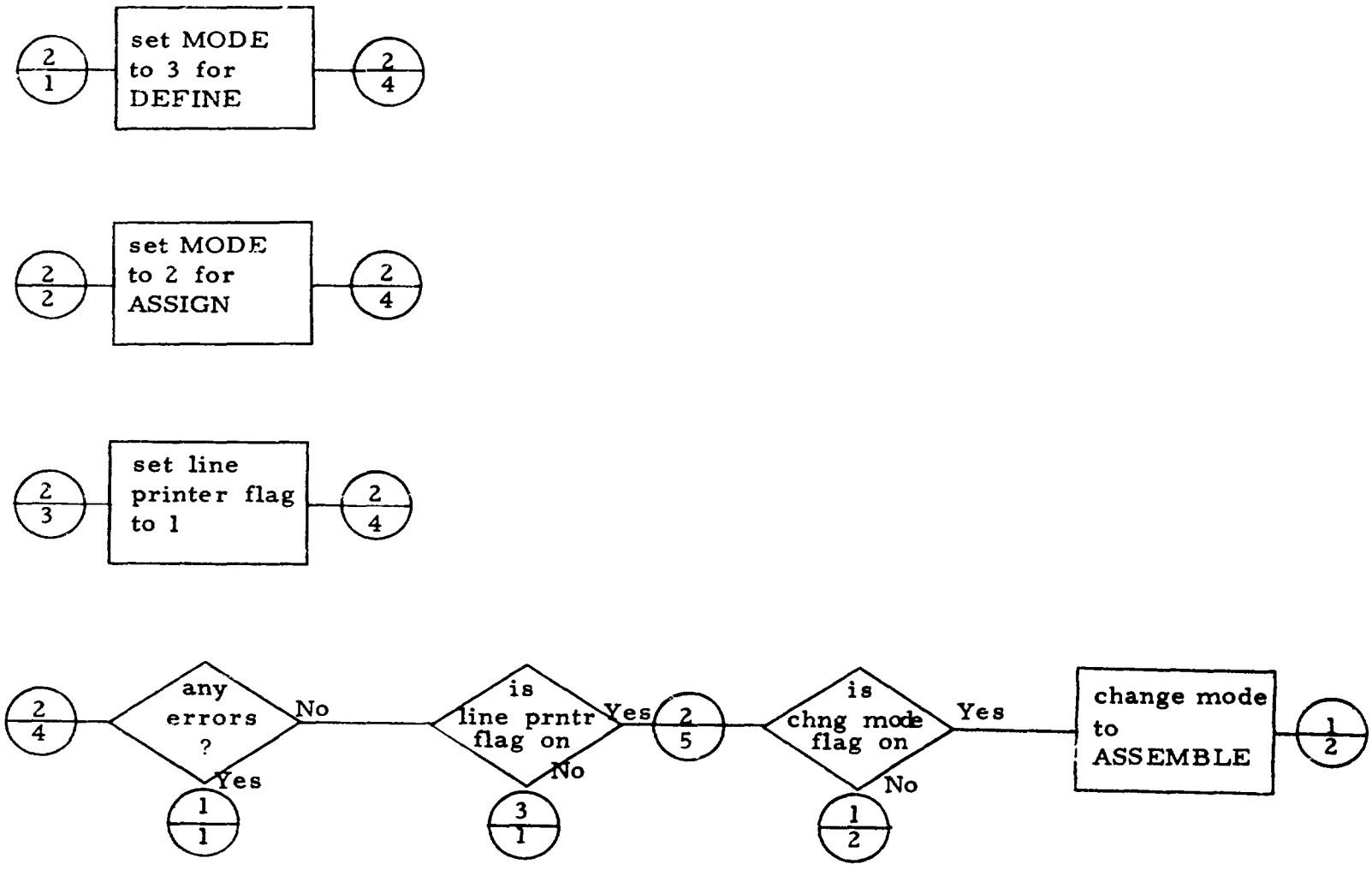


Figure 6-1 (Cont'd)

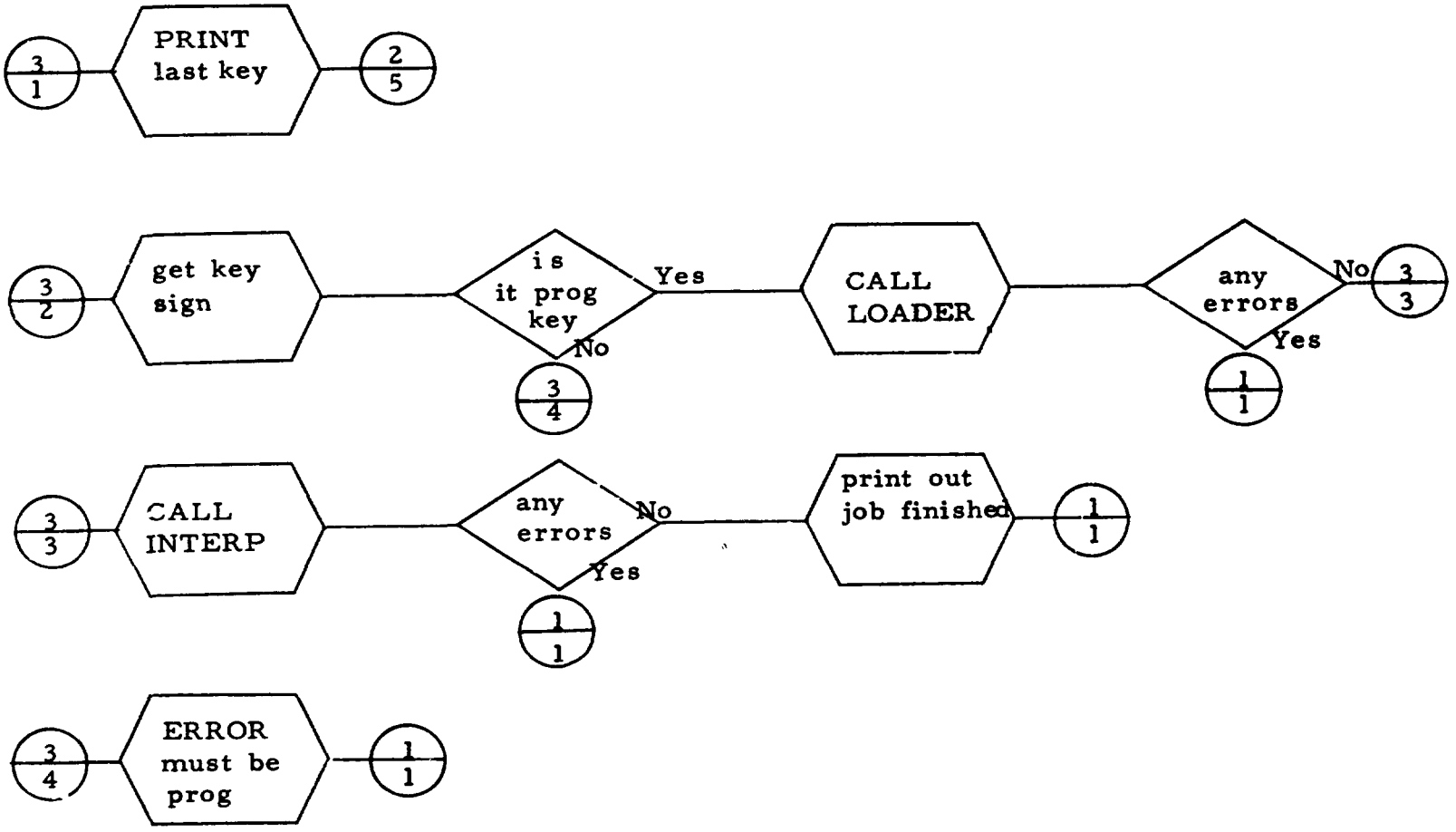


Figure 6-1 (Cont'd)

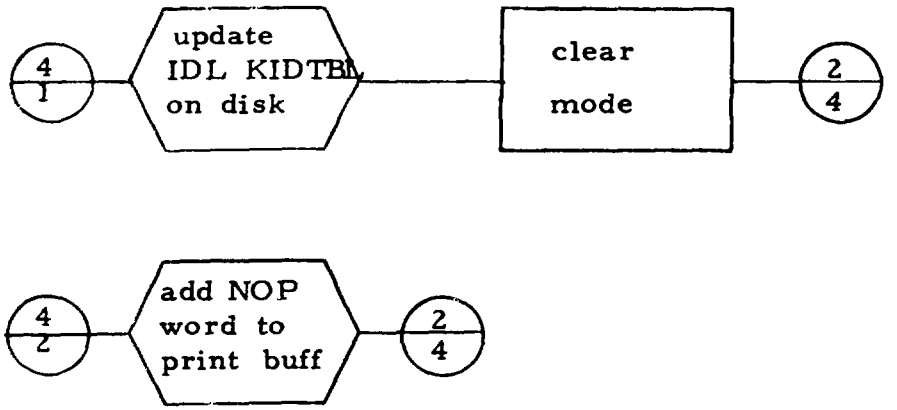


Figure 6-1 (Cont'd)

SECTION 7

IDL ASSEMBLER

7.1 GENERAL

The Assembler module is logically broken into 3 units; DEFINE, ASSIGN and ASSEMBLE. The DEFINE unit is responsible for setting up a new program format, the ASSIGN is responsible for assignment of the storage requirements, and the ASSEMBLE routine has the responsibility for building the Command structure. The following sections discuss in detail the function of each of the units of the Assembler. This discussion is followed by a complete detailed set of flow charts for the Assembler. (Figure 7-1 through 7-5).

7.2 DEFINE MODE

When the mode is set to 3 the Assembler jumps into the DEFINE routine. Using the key signature as an index, the entry in the identification table of the key to be defined is set to type 6 or "Program Key". Next the routine clears the counters and buffers which are used in the Assembly process. Finally it sets up the identifiers for each of the Loadable format packets. Control is passed back to the Executive to await another key signature.

7.3 ASSIGN MODE

The ASSIGN routine is entered only after an ASSIGN key has been received by the Executive and the mode set to 2. The ASSIGN routine requires a proper sequence of key signature inputs or an error will be generated. This sequence begins with an undefined key, which will become the name or label for the storage operand. The next entry must be the dimension of the operand or array. This dimension will be considered to be a decimal number. Following the entry of the

number key (s) for the dimension of the array, a "Delimiter" key (0536) must be entered. The Assembler provides a feature whereby the contents of the storage area may be specified by the user. This is accomplished using the Oct (0505) and Dec (0504) keys, which are followed by any of the number keys from overlay 5. Finally an "End" key must be entered to terminate the assignment of the operand.

7.3.1 Detailed Description

Upon entry to the ASSIGN routine the first key signature is checked to be sure it is an undefined (type 0) key. After an undefined key is received, several flags are cleared and the key signature is put in the defined storage packet. Control then returns to the Executive for another key signature.

The second key entered will be one of the number keys from overlay 5. A call will be made to the constant generating routine (GENCON). The constant generating routine will form the proper decimal value for those number keys which preceed the "Delimiter". After the "Delimiter" is received the constant generated, which is the dimension of the storage area, is stored in the defined storage packet.

At this point two types of keys are legal, an "End" key or one of the "Assemble Constant" keys. If there is to be no presetting of storage locations an "End" key will be received. The "End" key signals an "End" to the storage assignment process. All that is left is to set the label key to its proper type (4 or 8). If the dimension was only one word the label key becomes a type 4 or register key. The label key becomes a type 8 or block if the dimension is greater than 1.

If there is to be presetting of an array with data an "Assemble Constant" key will be received after the "Delimiter". The "Assemble Constant" sets up a jump to GENCON to handle the number

keys which follow. The series of number keys are formed into constants by GENCON. Each time a constant is completed by a "Delimiter" it is stored sequentially in the defined storage packet. When the "End" key is received, the array dimension entry is made negative. That is, one is set in bit 15 to indicate to the Loader that preset storage follows. The remainder of the process is the same as for the non preset assignment ending described previously.

7.4 ASSEMBLE MODE

Once the mode of the Executive has been set to 1, all key signatures are passed to the Assemble routine of the Assembler. After each key signature is processed by the various subroutine within the Assemble module, control is passed back to the Executive to await the key signature. Sections 7.4.1 through 7.4.3 discuss in general terms how most command and jump keys are processed. There are however two special keys which do not conform to the general rule. The "Call" command key (Section 7.4.4) and the "End" command key (Section 7.4.5) will be discussed separately.

When a command or jump key is received it indicates to the Assemble routine that all information pertaining to the previous key is complete. The Assemble routine performs several operations upon the data of the previous key. However to make this discussion as clear as possible we shall begin our discussion at the point where the Assembler begins to process the new key. Reference should be made to the flow charts for complete detailed descriptions.

7.4.1 New Key Subroutine - NEWT

The key signature is stored in the Command packet. The type of the key being processed is checked for Command (type 2) or jump (type 3).

If the key is Command, the syntax type will be stored after the Command entry in the Command packet. The syntax is used to check for special cases at this point. Commands which use syntax 6 must be handled differently so a flag SYN 6 is set if the syntax equals 6. The "End" key is a type 0 syntax so a check for that key is done here. See LABCHK (Section 7.4.5) for an explantion of the "End" key function. If an "End" key is found, a jump to LABCHK is executed. If not dealing with special keys the number of parameters for the syntax type is stored in PNUM, which is used as a counter. Several counters and buffer areas are cleared and control is returned to the Executive for another key which will be a parameter.

Had the key received been a type 3 key, a test would have been made to determine whether the key was a jump key or "Label Follows" key. In both cases the parameter counter PNUM would have been set to 1. However, the flag jump will be set to 1 for jumps and 2 for "Label Follows" key. Here again counters and buffers are cleared and control passed back to the Executive.

7.4.2 Parameter Inputs

As the parameters come in for processing, the jump flag is checked. If the flag is set, the parameter key is stored in JTAB. JTAB is a buffer used to hold the targets of all jumps. This table will be used later to determine if all jump targets have been given a label in the transfer packet. The target key is stored immediately after the jump Command key in the Command packet.

The parameters may come to the Assembler in two forms. They may be a key signature which has been previously assigned or they may be series of keys to make up a constant. The handling of the parameters in both cases is the same. However,

a series of type F key signatures must first be made into the constant they represent.

When a type F key is recognized a return jump to the constant generating routine (GENCON) is performed. All type F keys are sent to GENCON until the constant has been completed. The completed constant is then stored in the constant packet. A pointer to the constant will be used as the parameter. The pointer will be in the form of a one word index count. The index word is given the key type of F.

Parameters are checked out for type. When a user uses a key he assigned earlier, in the ASSIGN mode, the key type will be 0. The key identification table is not updated until the Assembly is finished. Therefore it is necessary to check the buffer which holds the newly assigned keys for the correct key type. Once Loaded this new key type is appended to the parameters key signature. Finally all parameter key signatures are stored in a six word buffer called PARAM.

7.4.3 Syntax Checking

The syntax of the parameters must be checked for legality. The checking is performed when the Assembler detects a new Command key is to be processed. To accomplish the checking the key is obtained from each key signature in PARAM. The key types are packed into a two word buffer called SYNBUF. The first word of SYNBUF holds the key types from the first 4 words in PARAM. The second word contains the last two. There are individual subroutines to check syntax type however they are all basically the same. (Refer to flow charts). Each subroutine searches its syntax table (Section 3) for an entry which matches SYNBUF. If no match is found, the appropriate error message is generated and the assembly aborted.

Exit from all syntax checking subroutines is to another subroutine called PACK. It is the function of PACK to move all the parameters from PARAM to the Command packet. After this function has been performed the Assembler is ready to start a new Command.

7.4.4 Call Command

The "Call" key is checked for by the syntax checking subroutine for syntax 5. When a "Call" key is found a jump to the subroutine "Call" is performed. The program which is to be called will act as a subroutine to the program being compiled.

The "Call" routine uses the parameter, which must be a program name, as an index into the key identification table. The disk address of the called program is obtained from the key identification table. The first 3 packets are read from the disk. The storage packet of the called program is added to the storage packet of the program being compiled. Next, the constant and transfer vector packets are added in their entirety to the Command packet under construction. Using the address from the Loader data the Command packet of the called program is read into the Command packet being compiled.

The last word in any Command packet is the signature of the "End" key. To indicate that the "End" key now represents the end of a subroutine, the most significant 4 bits of the word are set to a hexidecimal 8. The index into the Command packet is updated to reflect the additions. Control passes to NEWT to begin processing a new key.

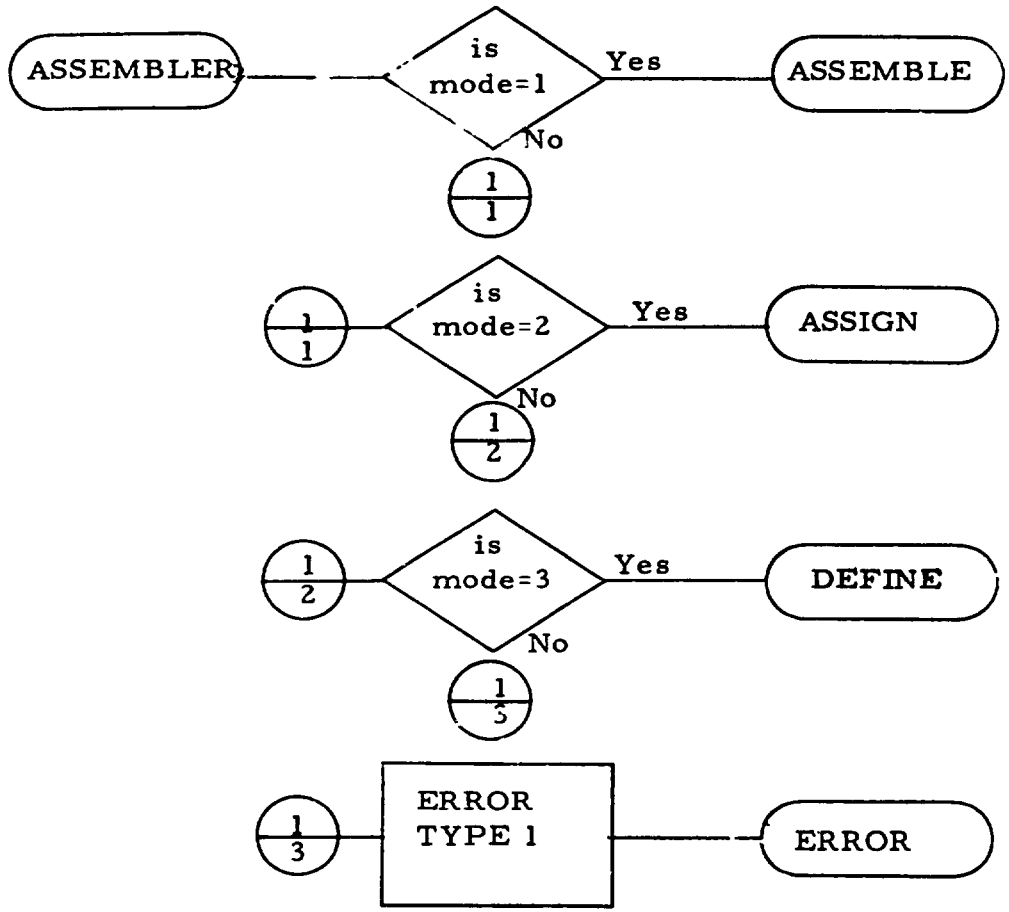
7.4.5 End Command

When an "End" key is encountered by the Assemble routine a jump to LABCHK is performed. LABCHK performs a check

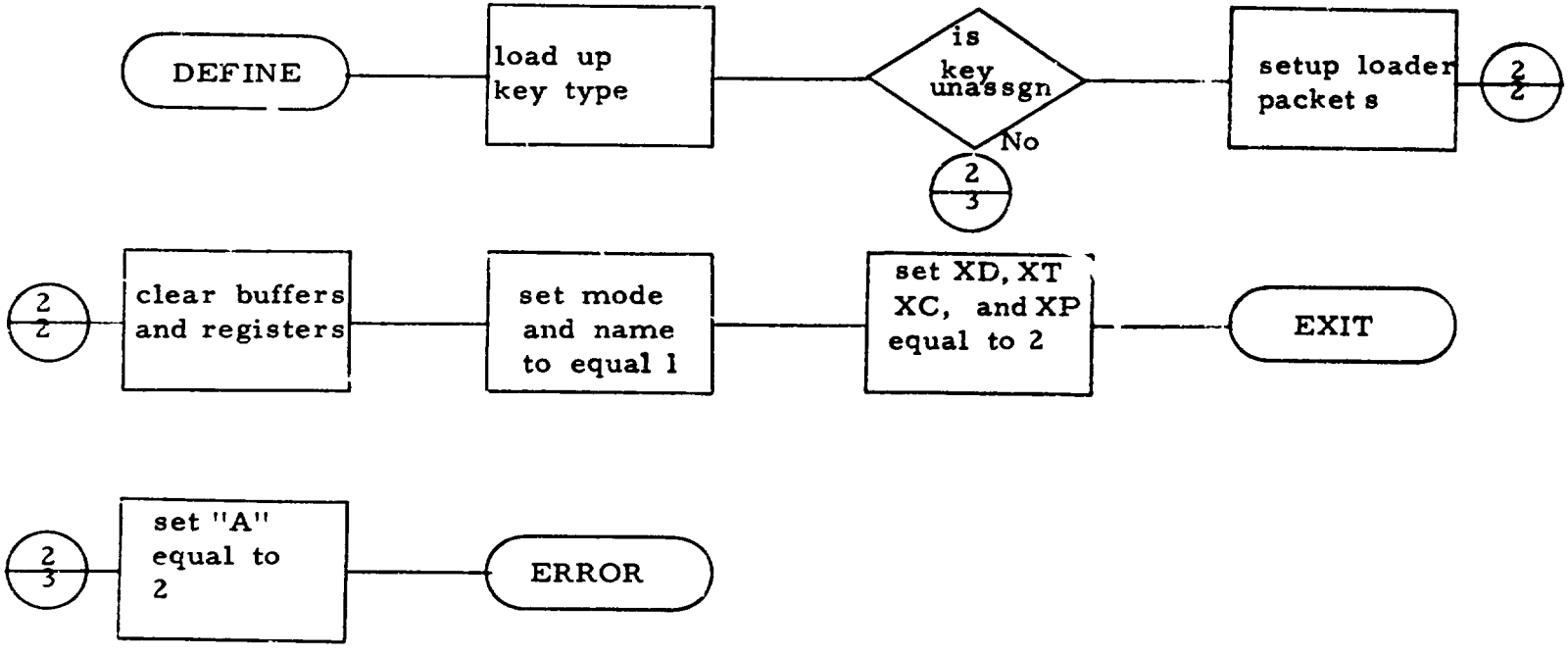
to be sure each jump target stored in JTAB has a corresponding label in the transfer vector packet. If a label is missing an error message will be generated. If all targets have labels, control goes to the subroutine THEEND.

THEEND builds the Loader data from the indices of each of the packets. Next the key identification table is updated. The information for the update is available from a buffer called NEWKID which holds all newly assigned keys. The entry in the key ID table for the program name must also have its key type changed to 6.

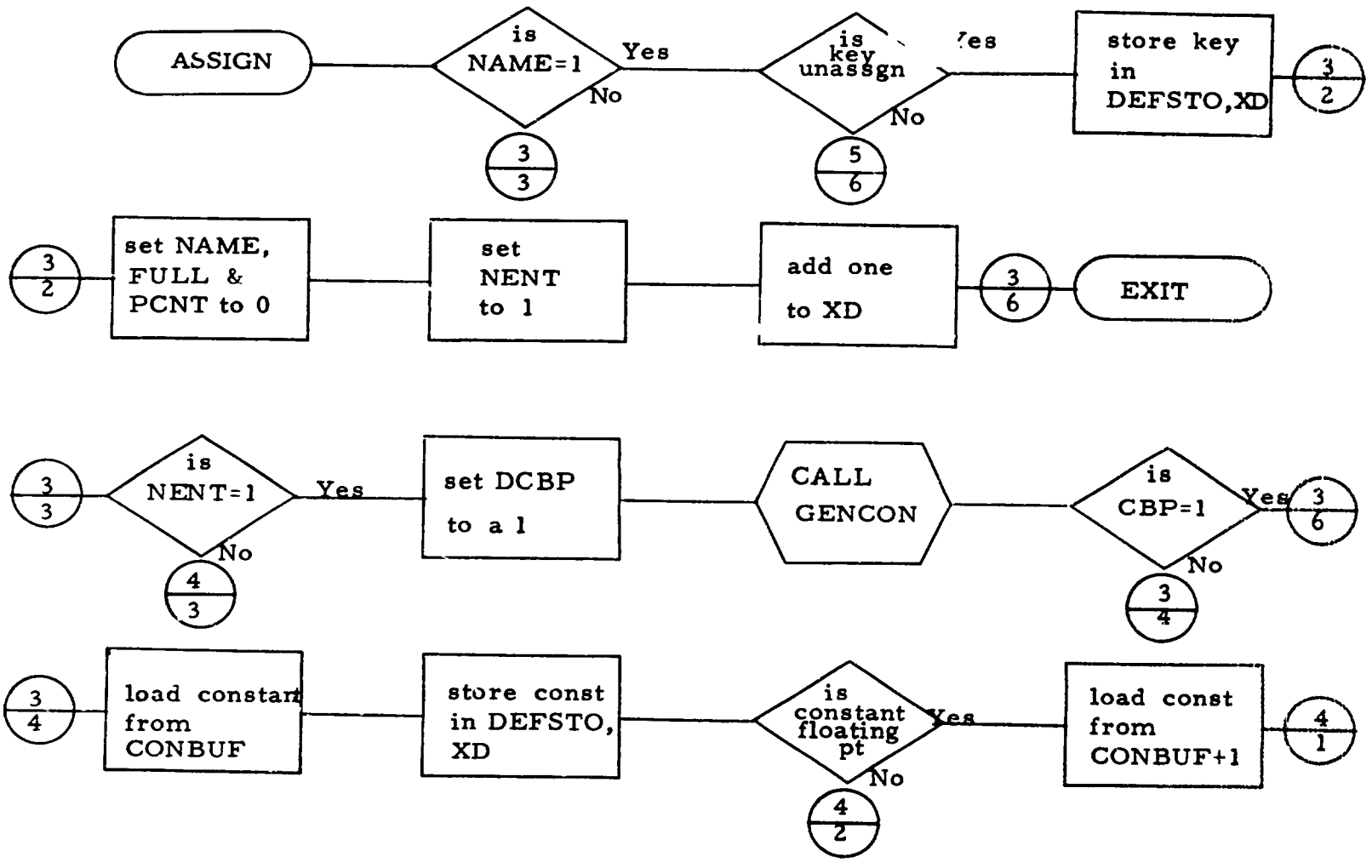
All that remains is to write the completed program on the disk. The first two entries in the key identification table contain disk information. The first word is the disk address of the key identification table. The second word is the next available sector address to be used for a program. The program is written starting at that sector. The next available section disk address is updated by the number of sectors used. The Assembler is now finished, the "A" register is cleared indicating no errors and control is returned to the Executive.



Assembler Inititation
Figure 7-1



Define Mode
Figure 7-2



Assign Mode
Figure 7-3

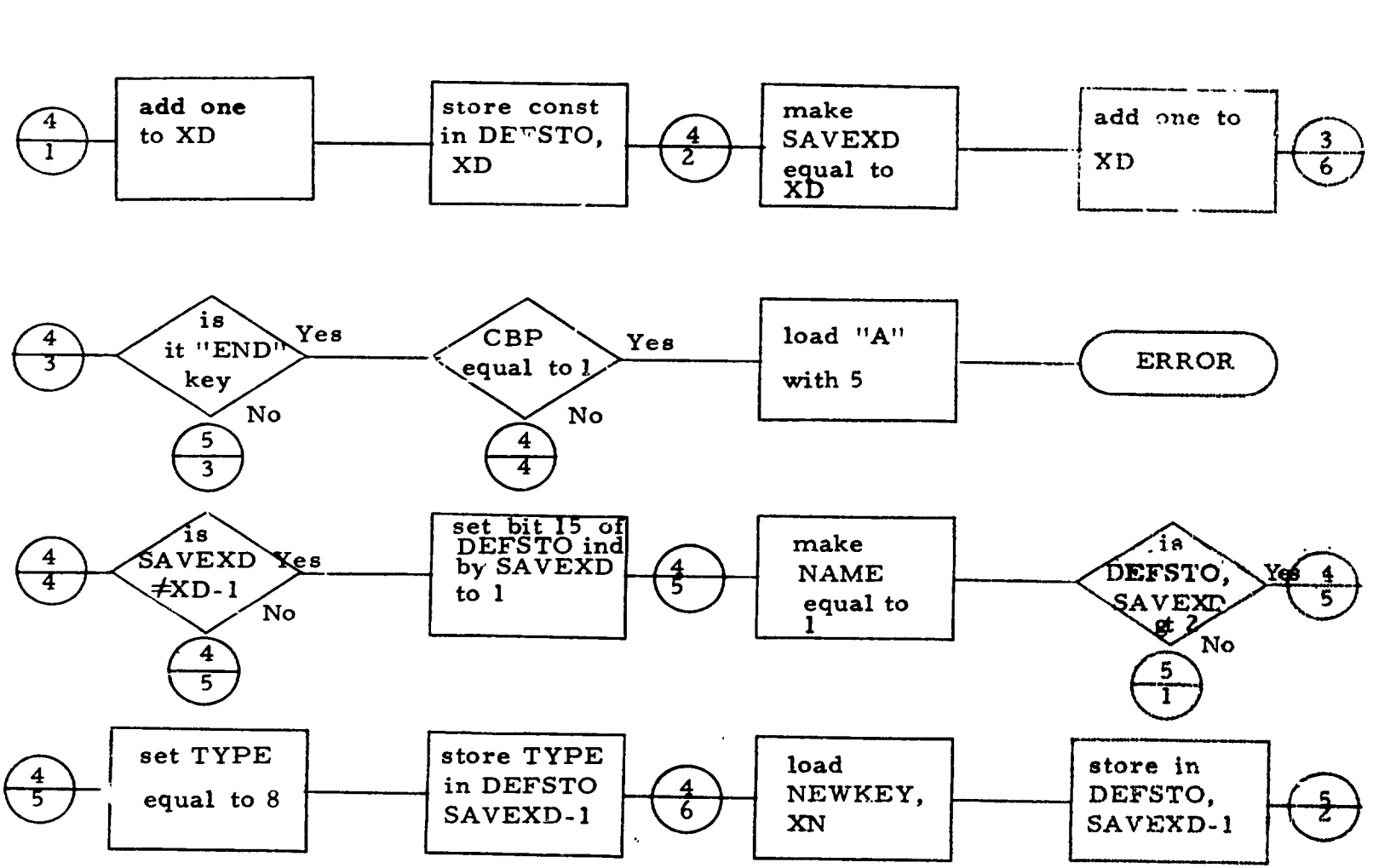


Figure 7-3 (Cont'd)

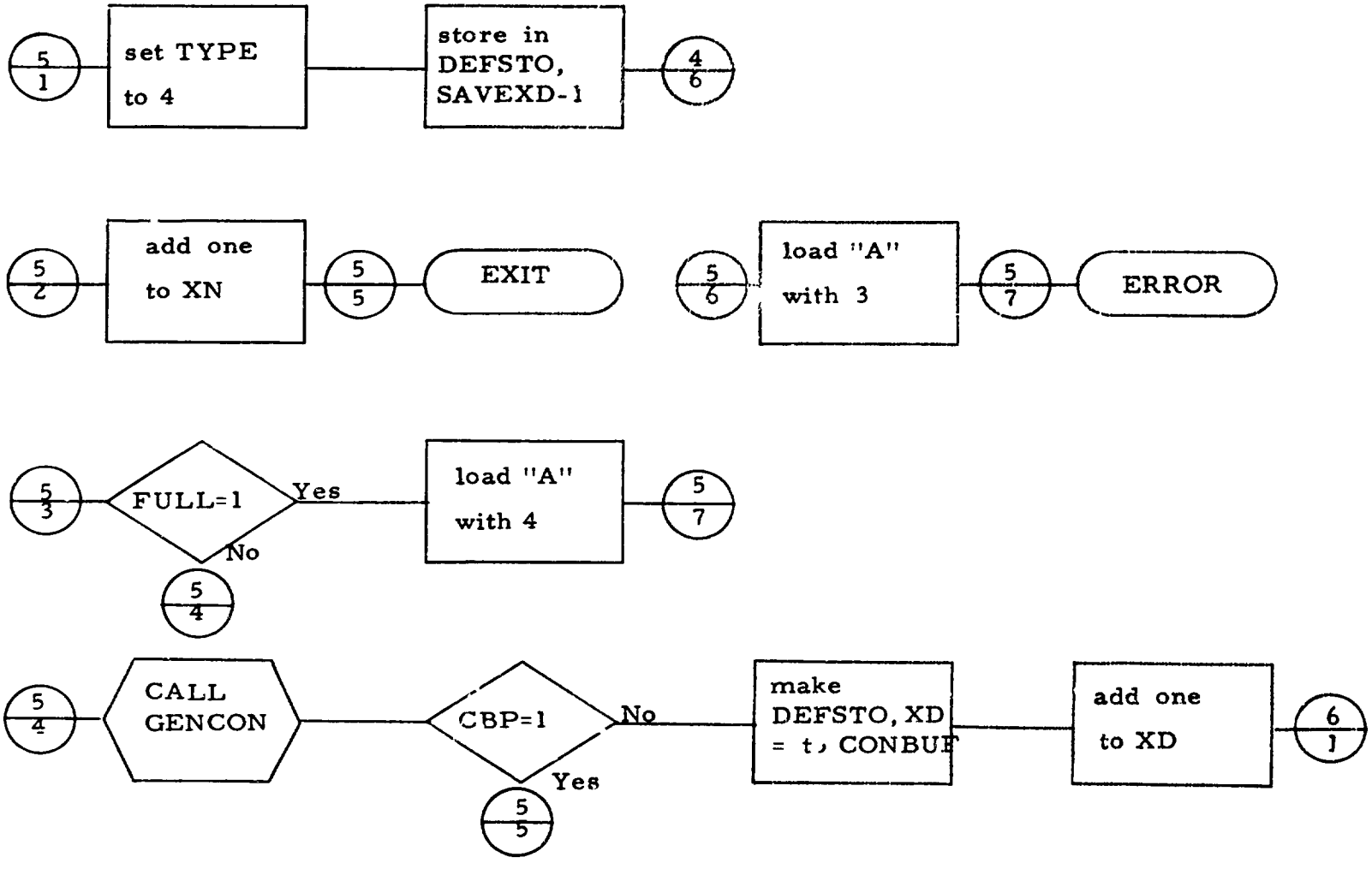


Figure 7-3 (Cont'd)

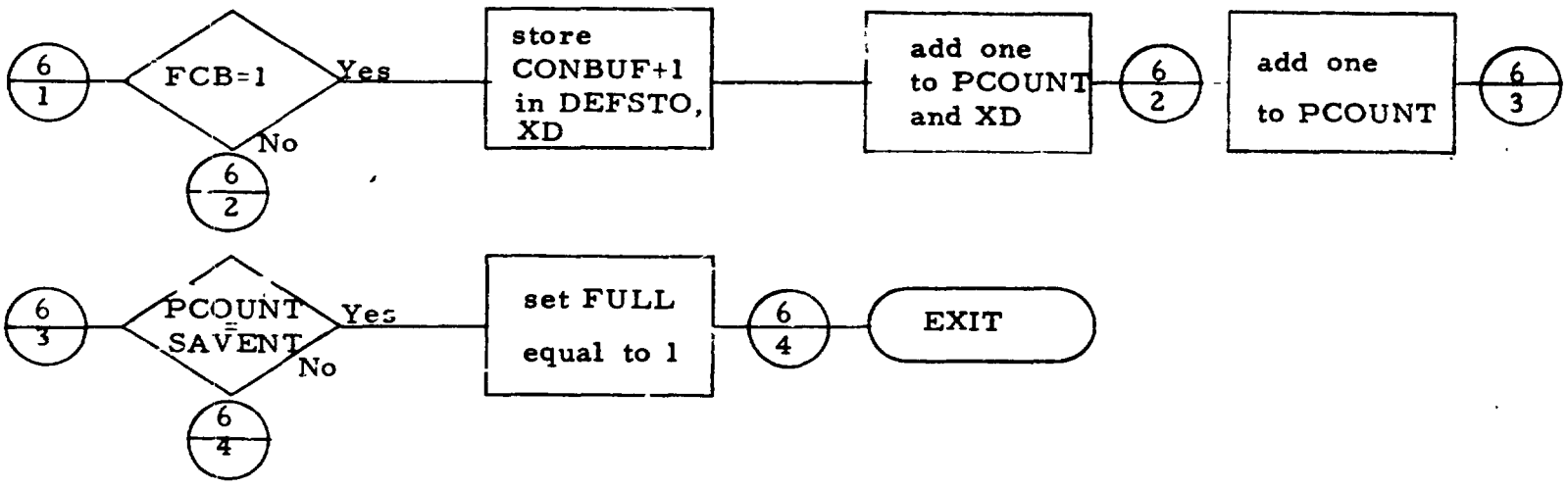
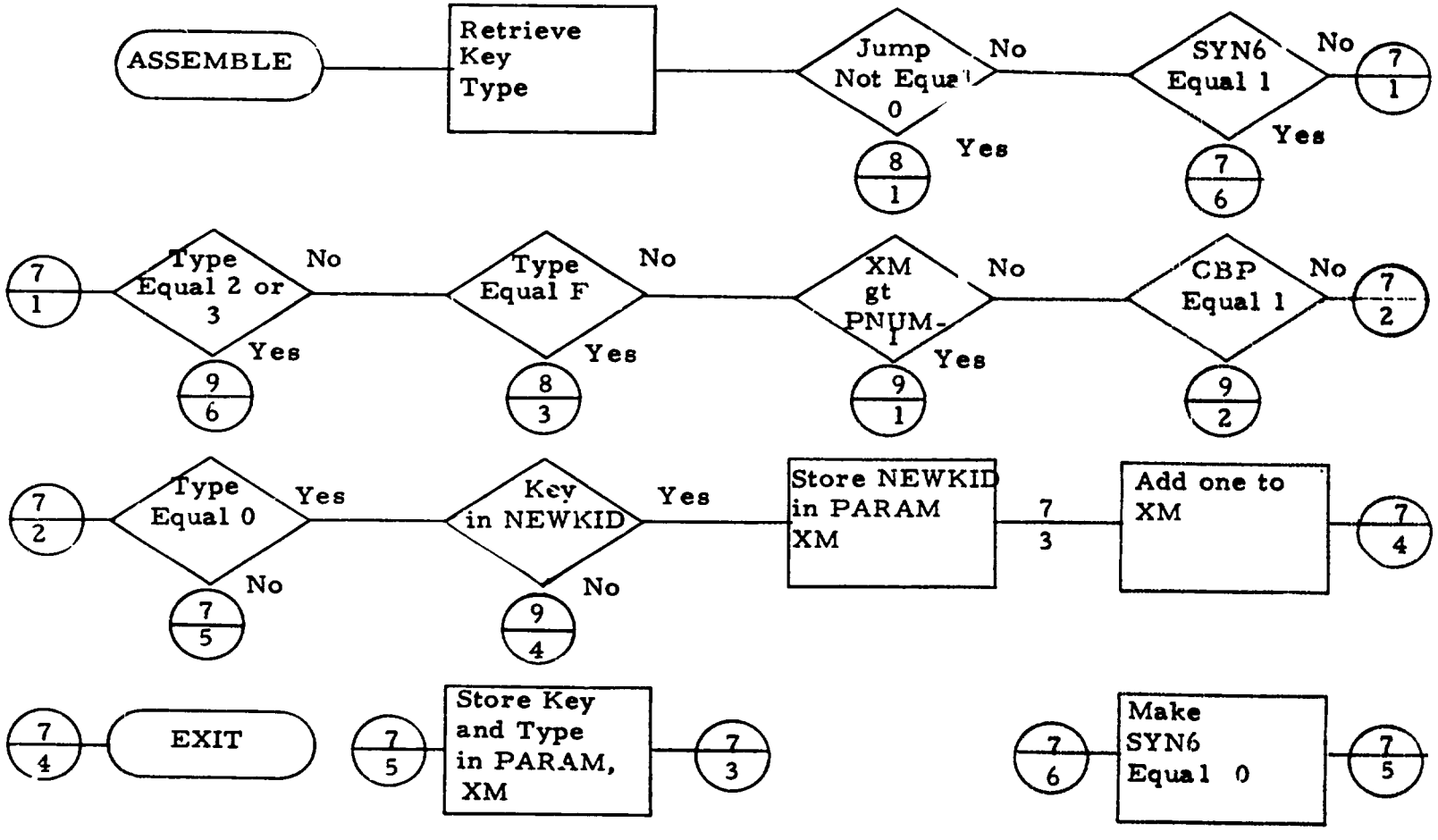


Figure 7-3 (Cont'd)



Assemble Mode

Figure 7-4

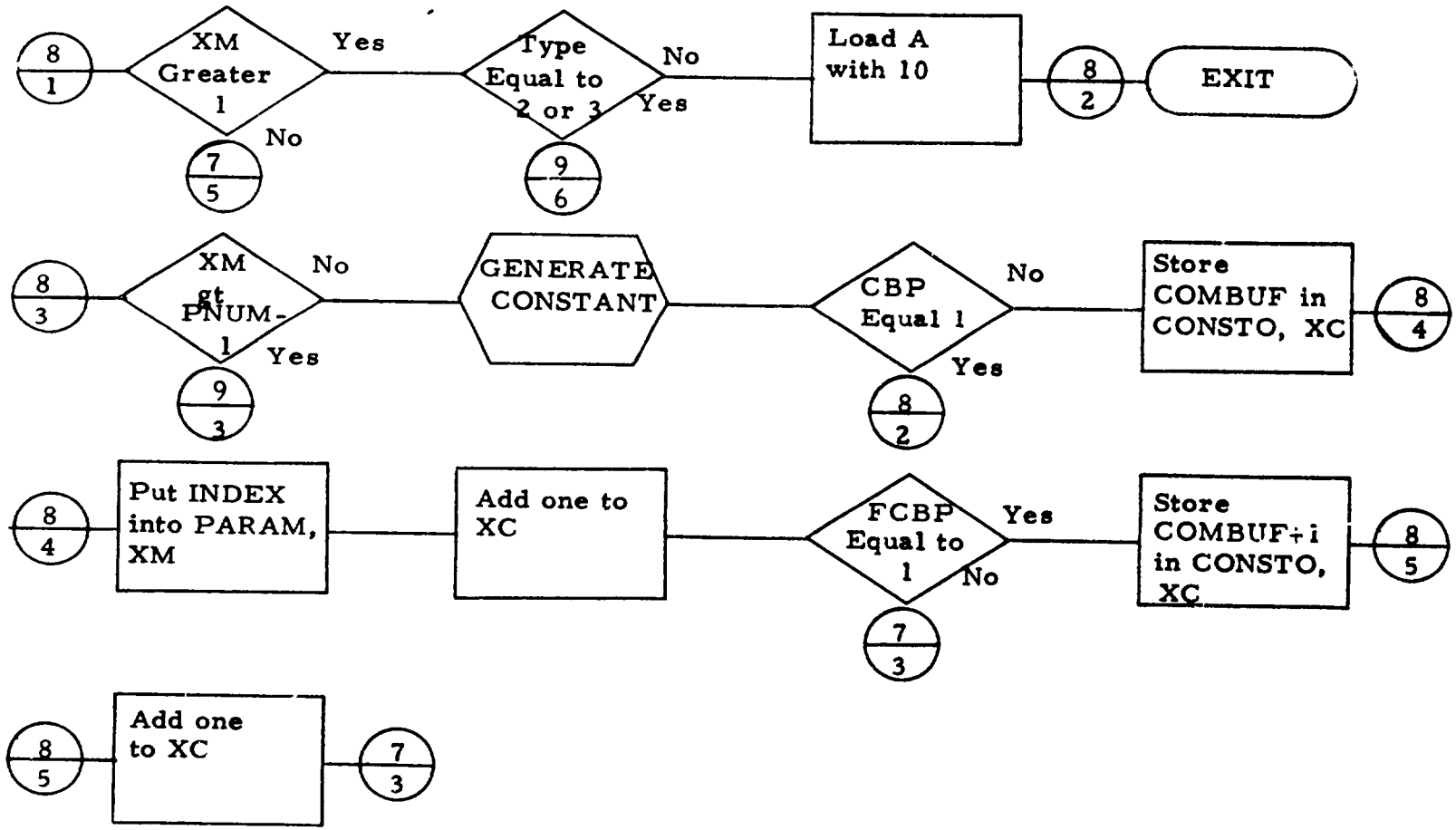


Figure 7-4 (Cont'd)

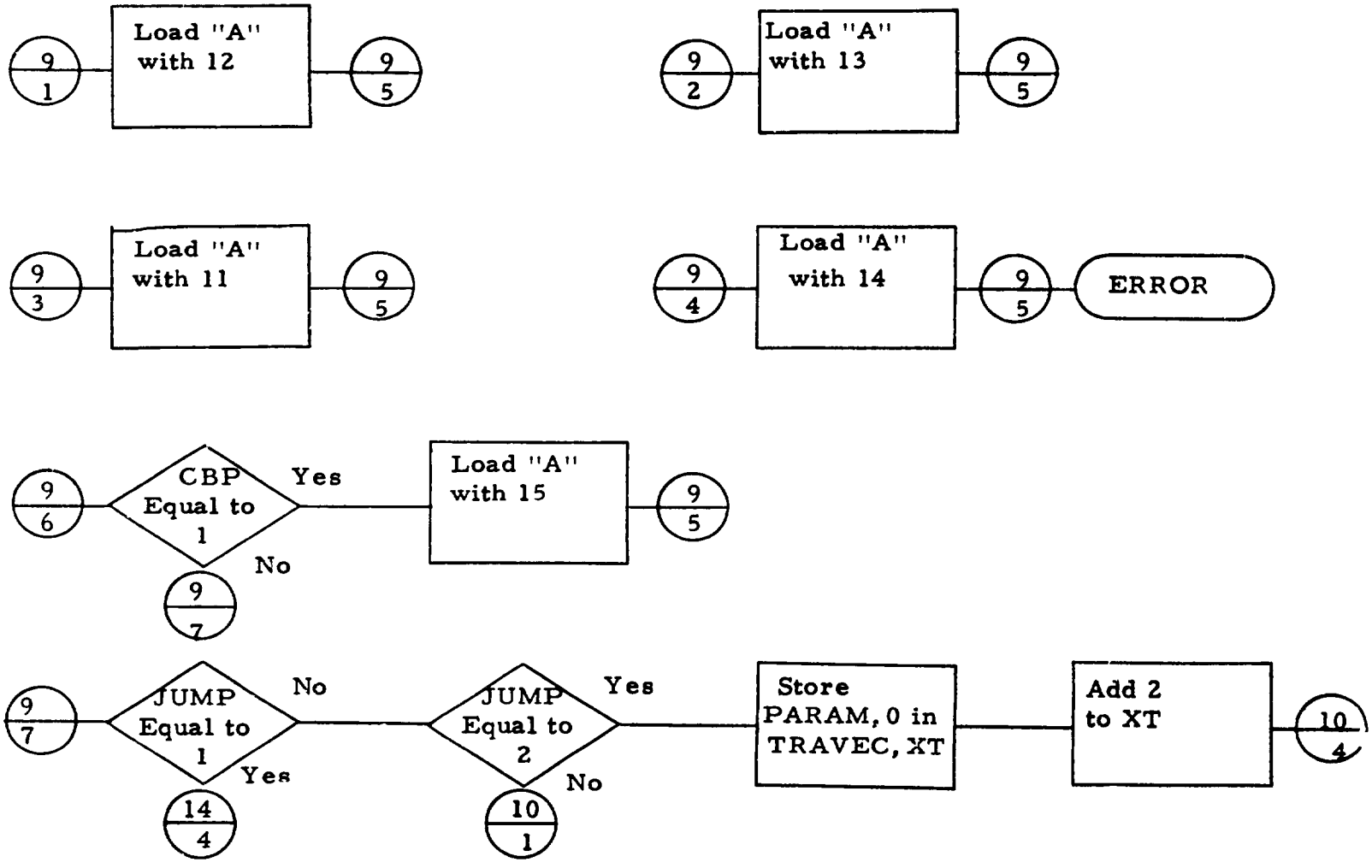


Figure 7-4 (Cont'd)

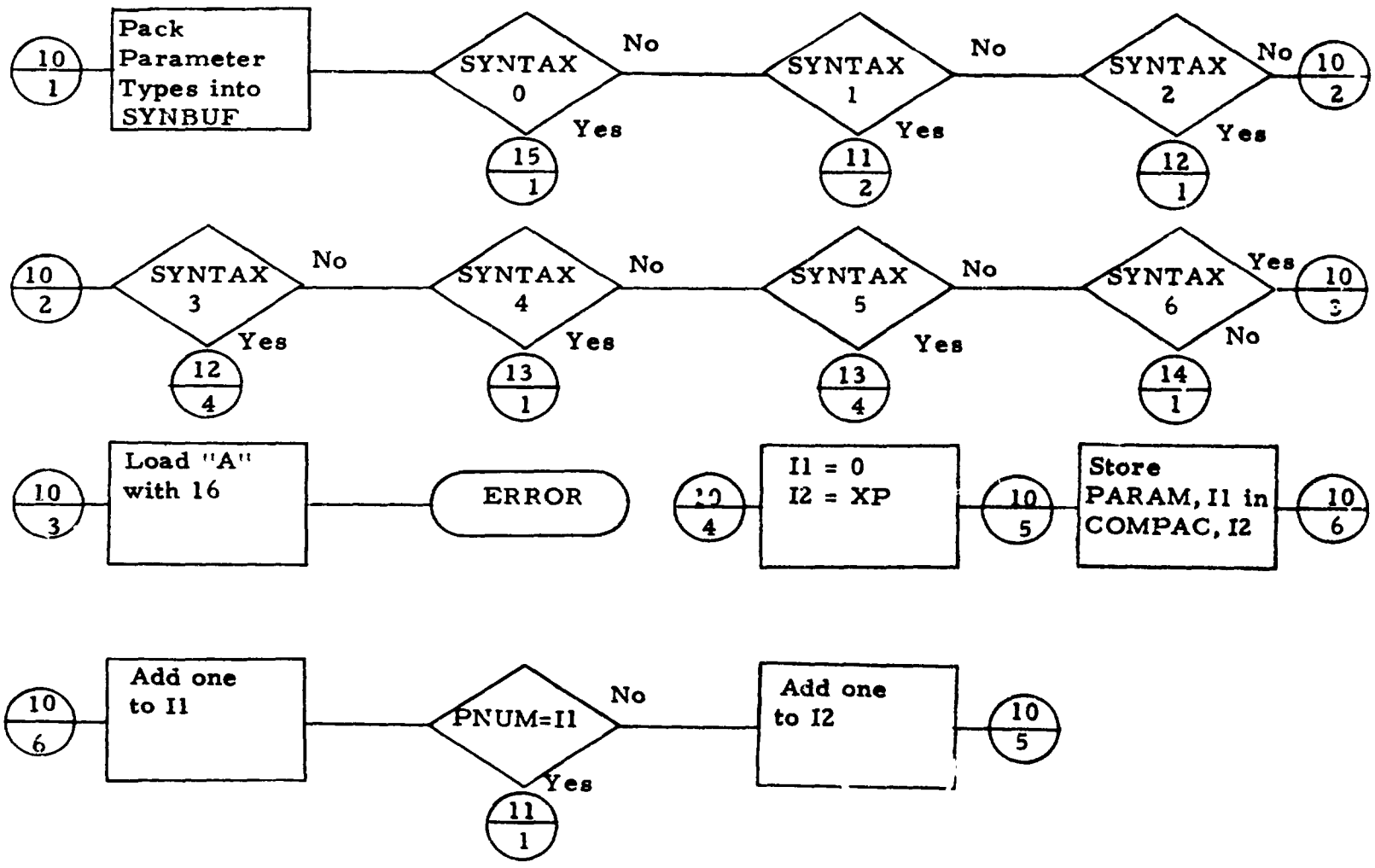


Figure 7-4 (Cont'd)

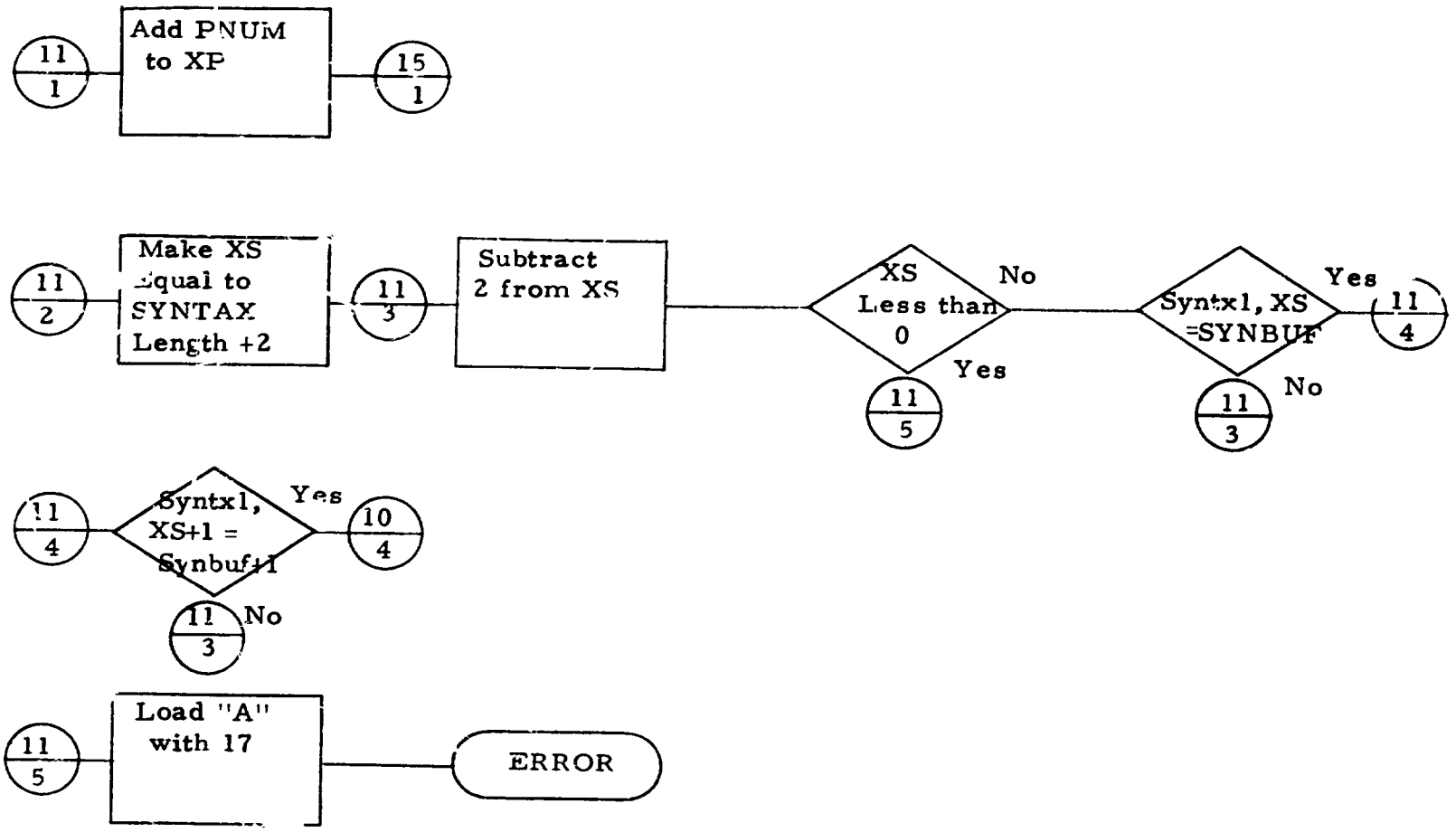


Figure 7-4 (Cont'd)

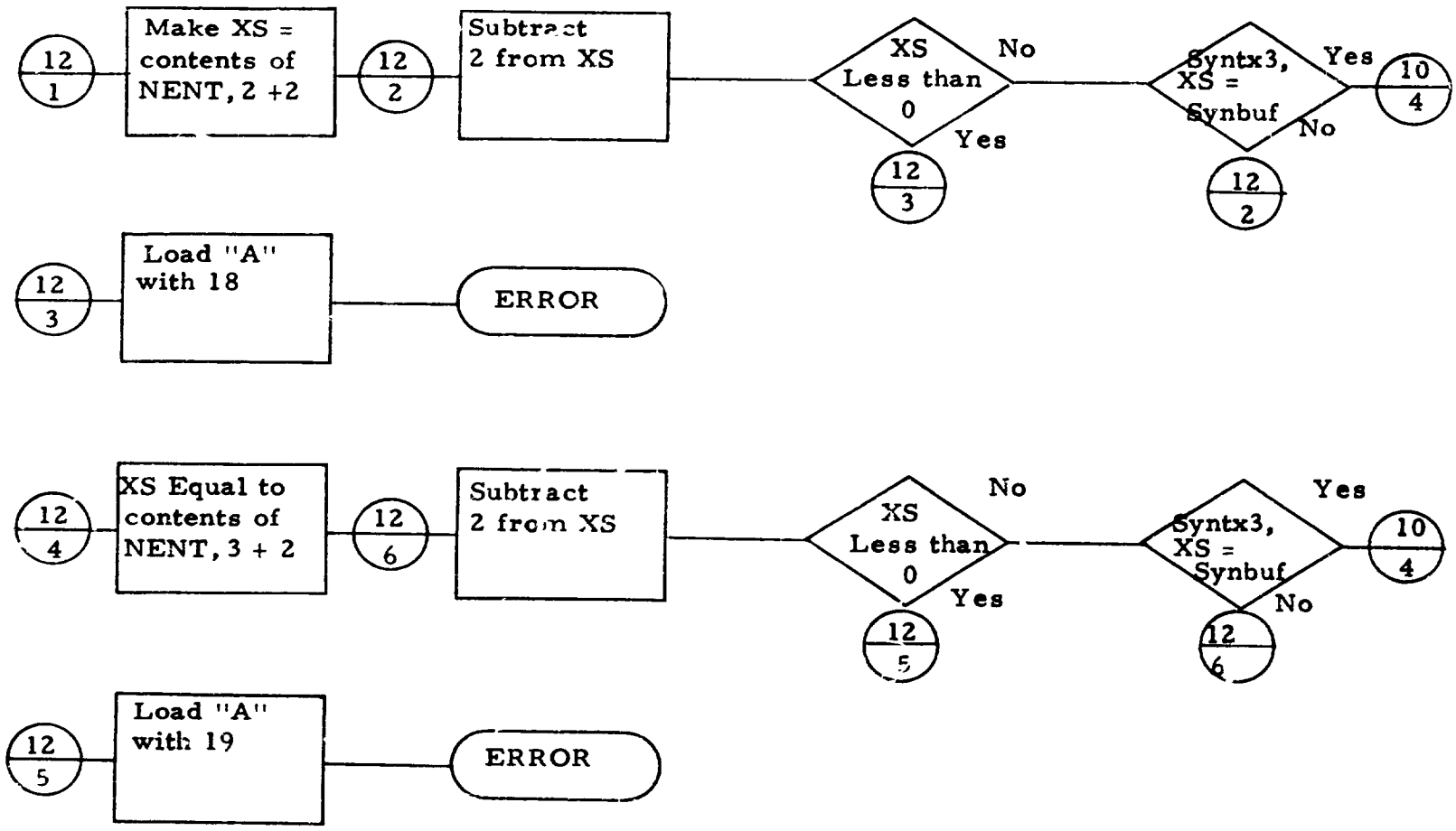


Figure 7-4 (Cont'd)

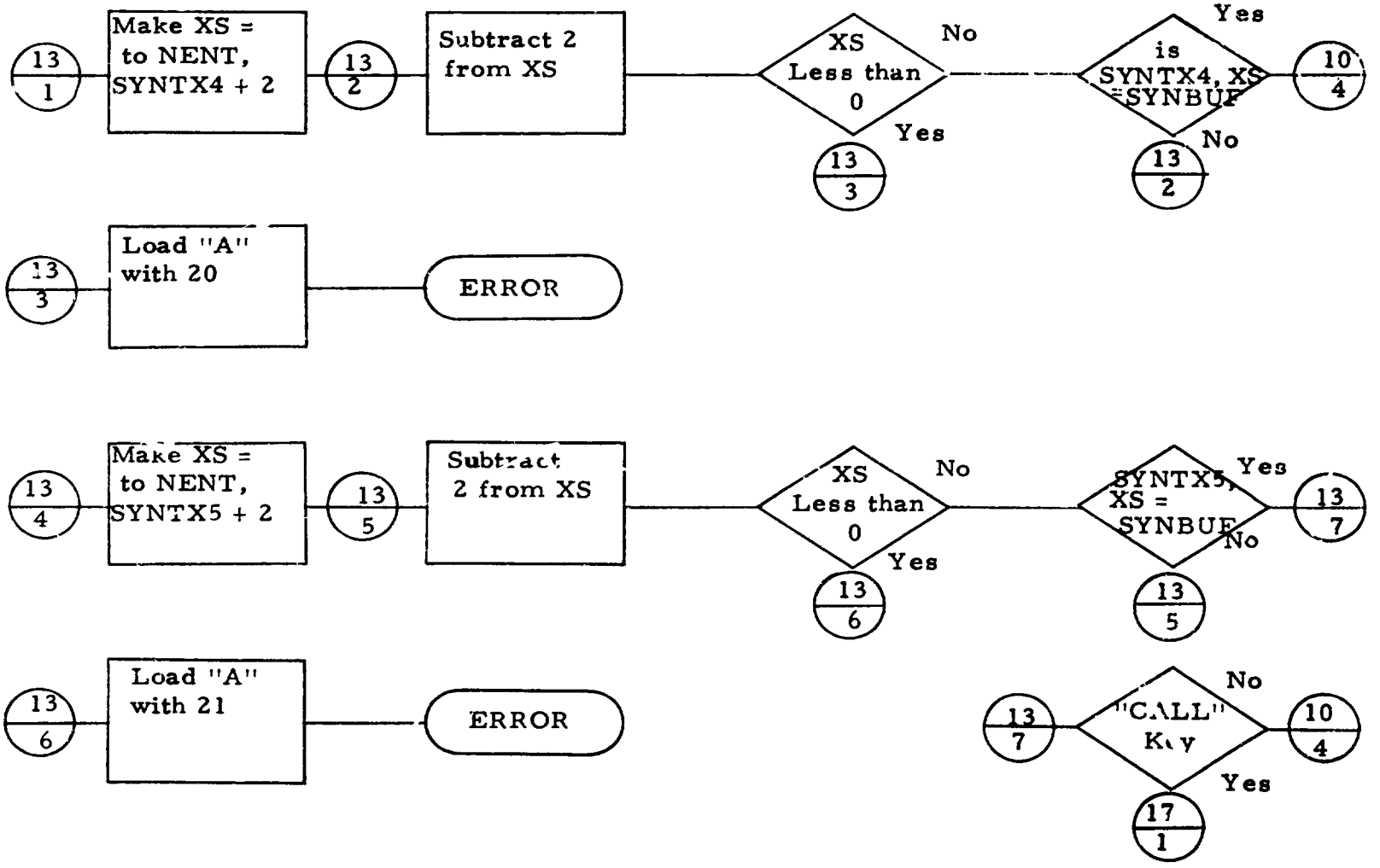


Figure 7-4 (Cont'd)

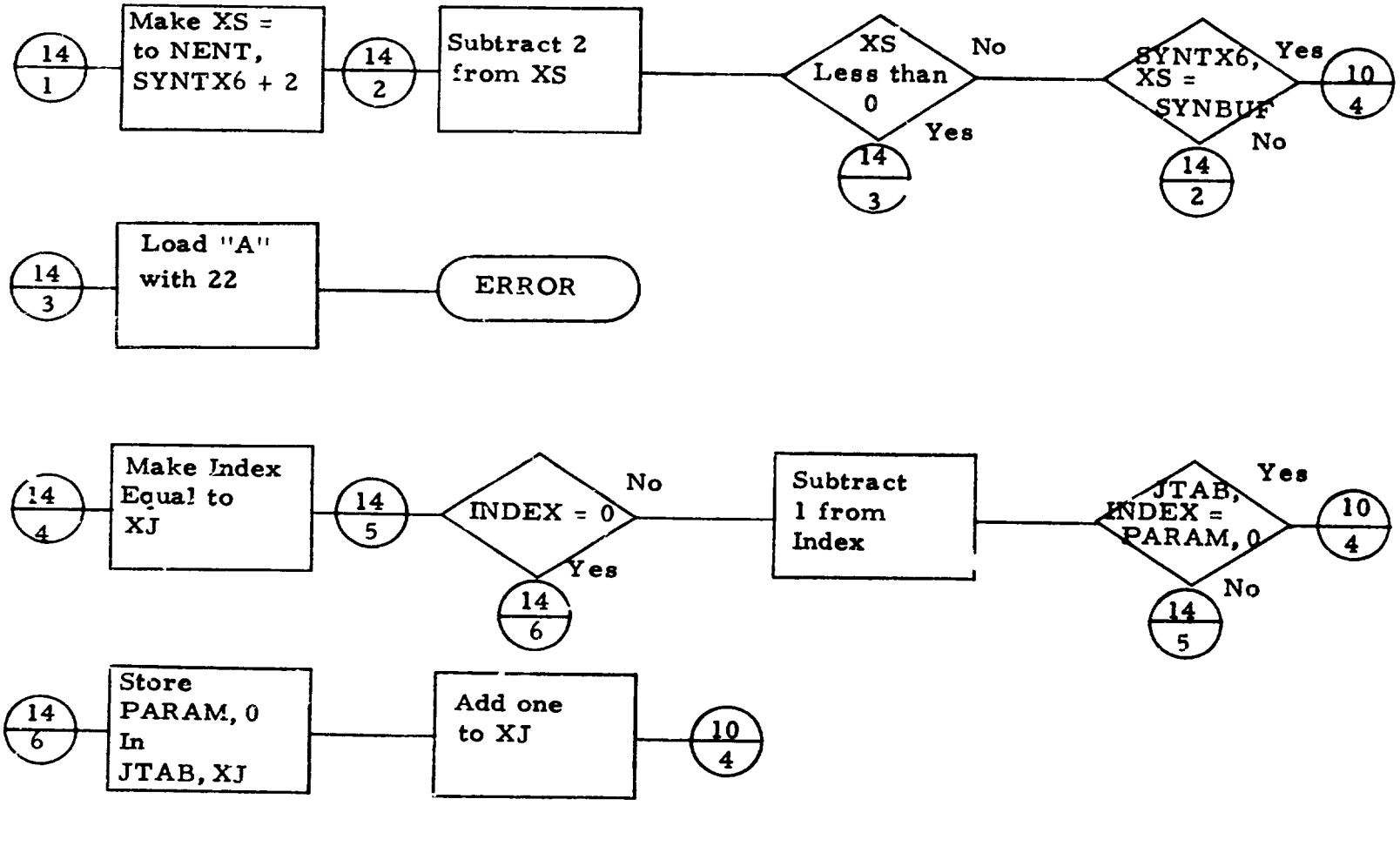


Figure 7-4 (Cont'd)

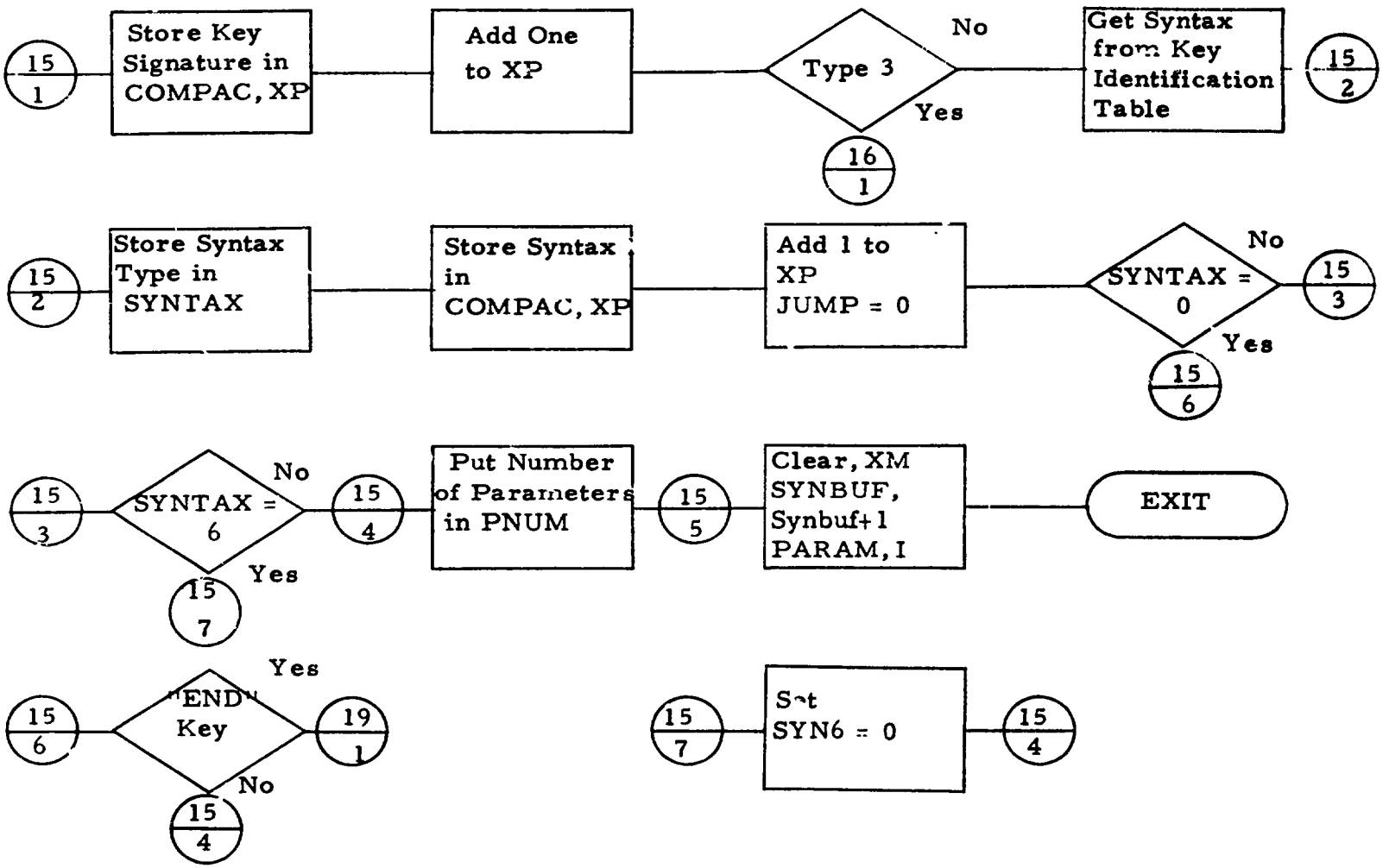


Figure 7-4 (Cont'd)

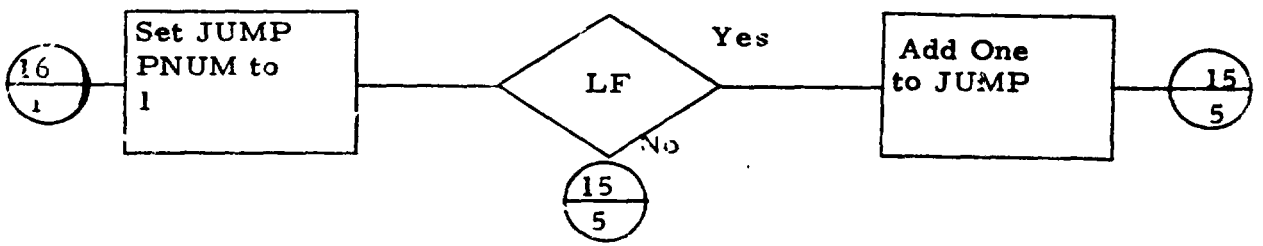


Figure 7-4 (Cont'd)

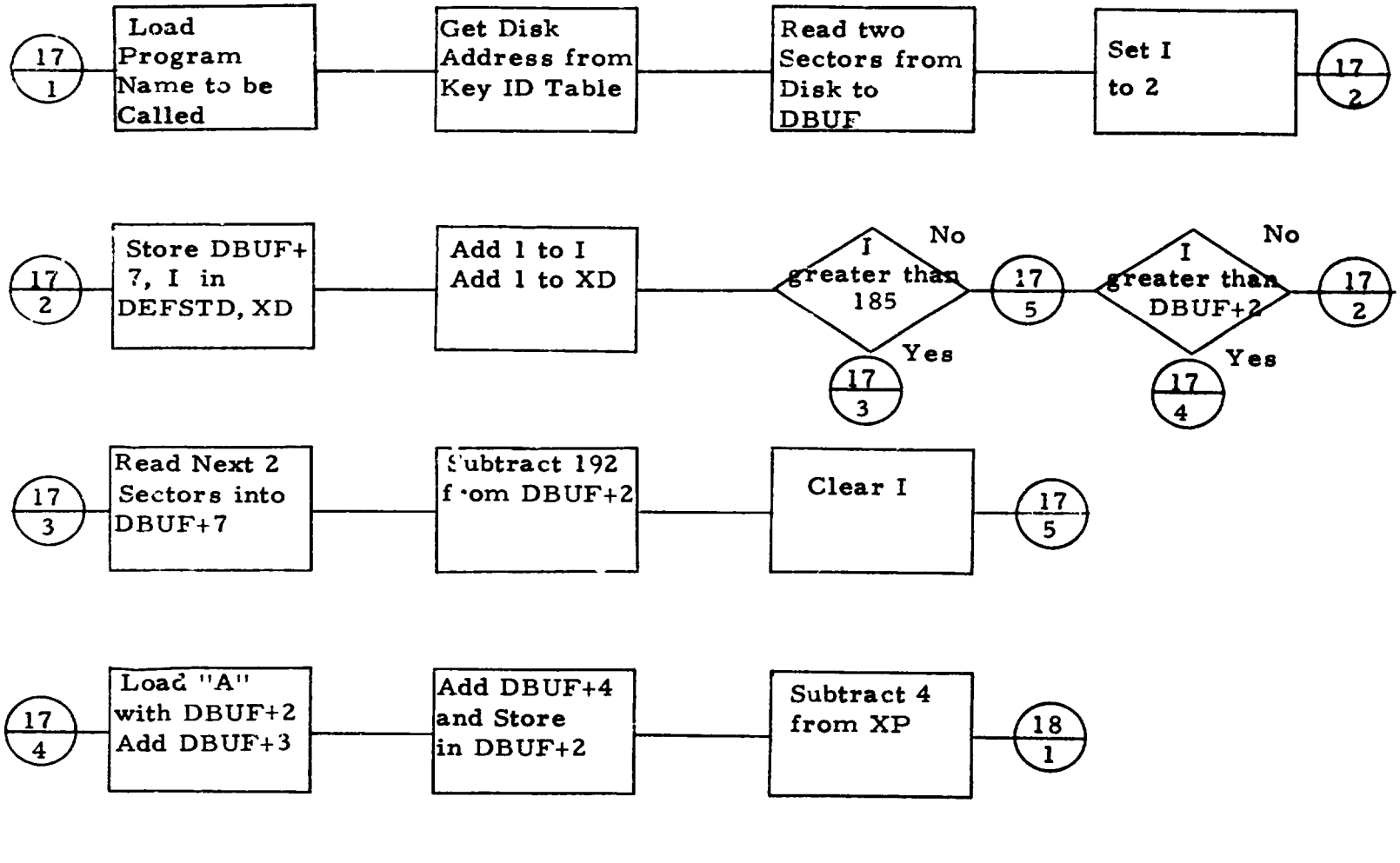


Figure 7-4 (Cont'd)

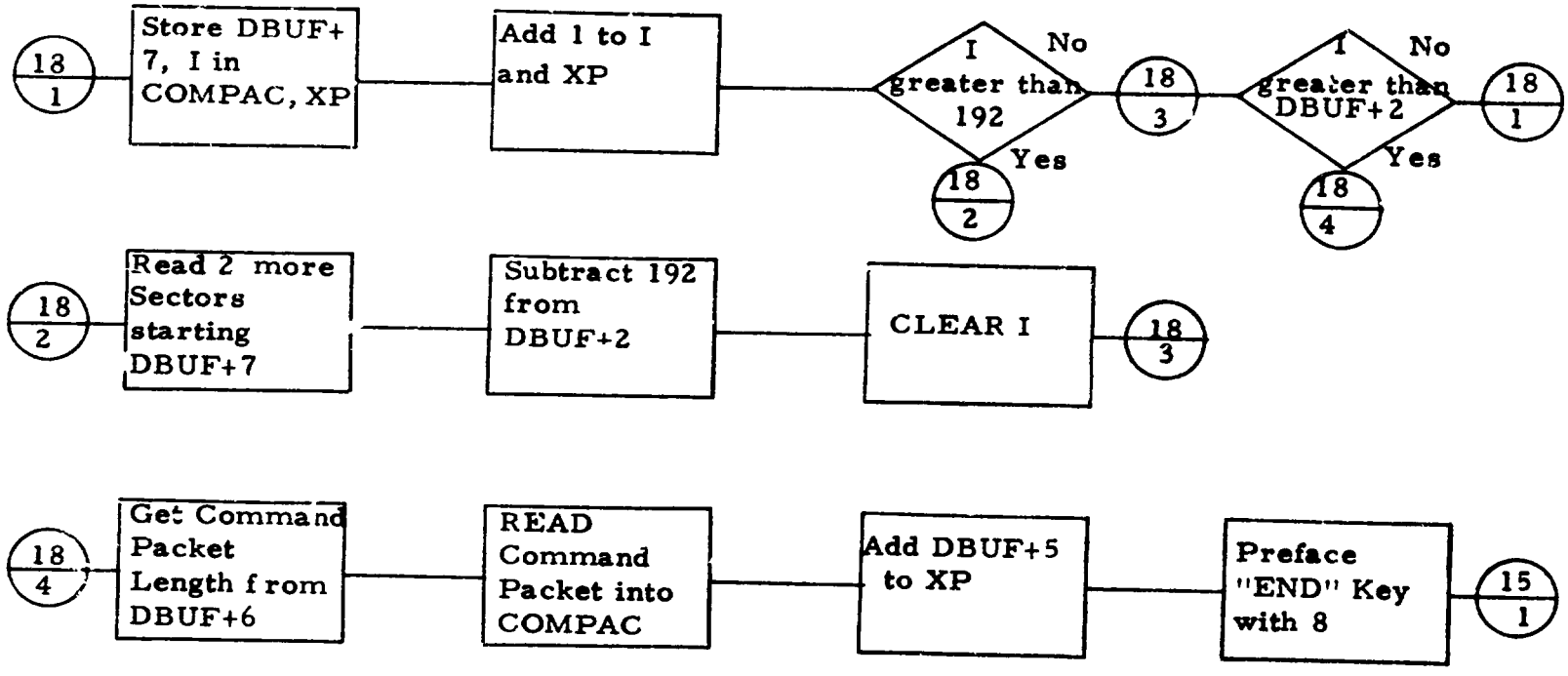


Figure 7-4 (Cont'd)

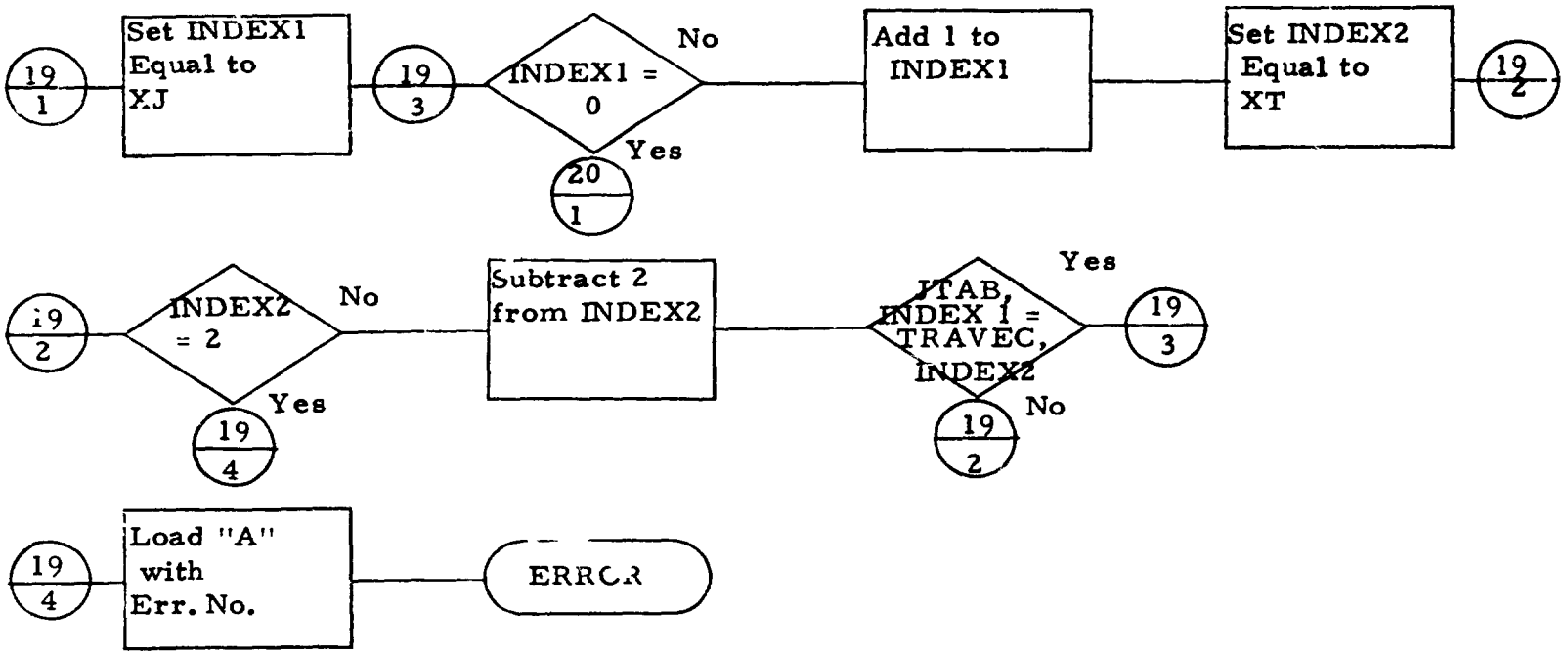


Figure 7-4 (Cont'd)

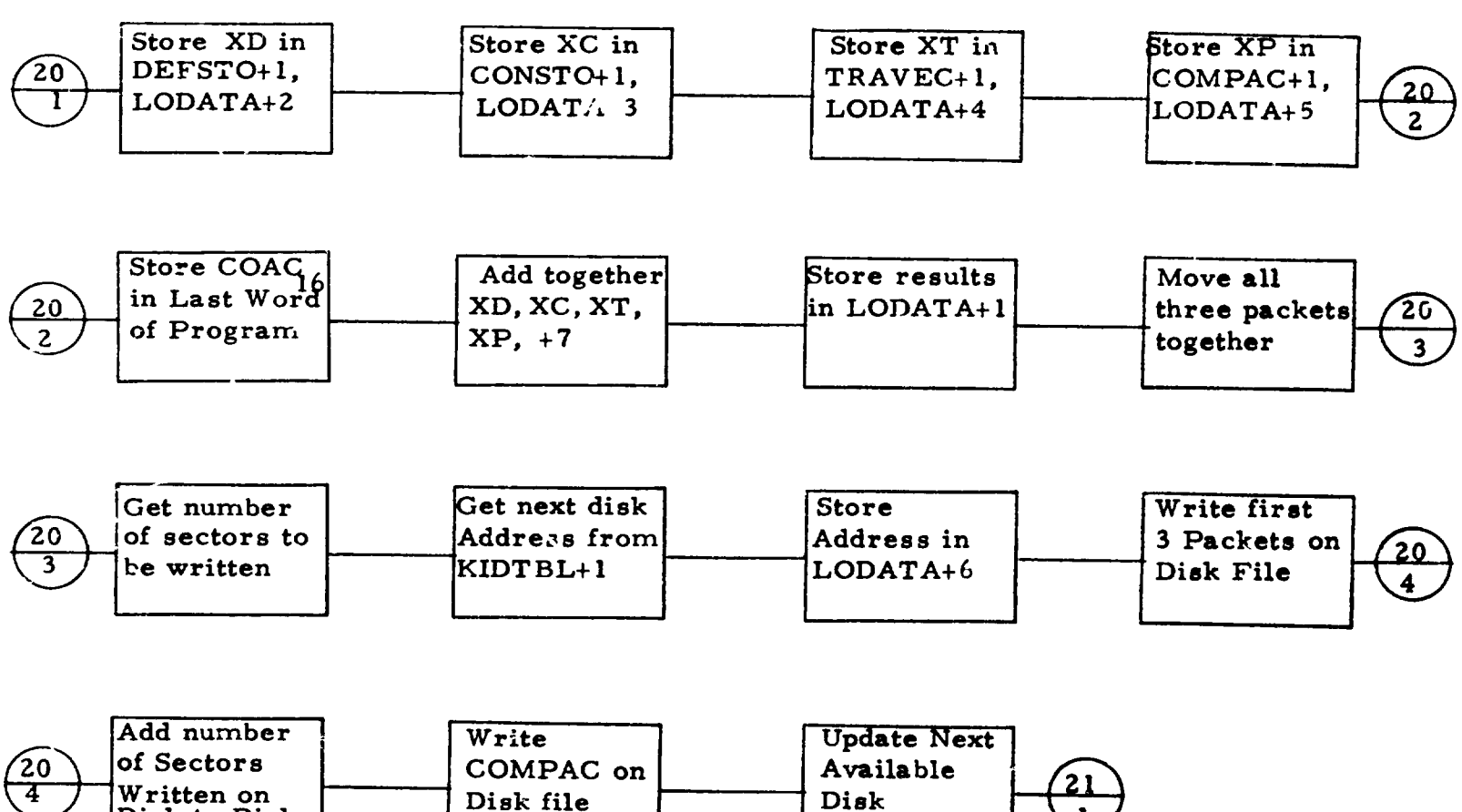


Figure 7-4 (Cont'd)

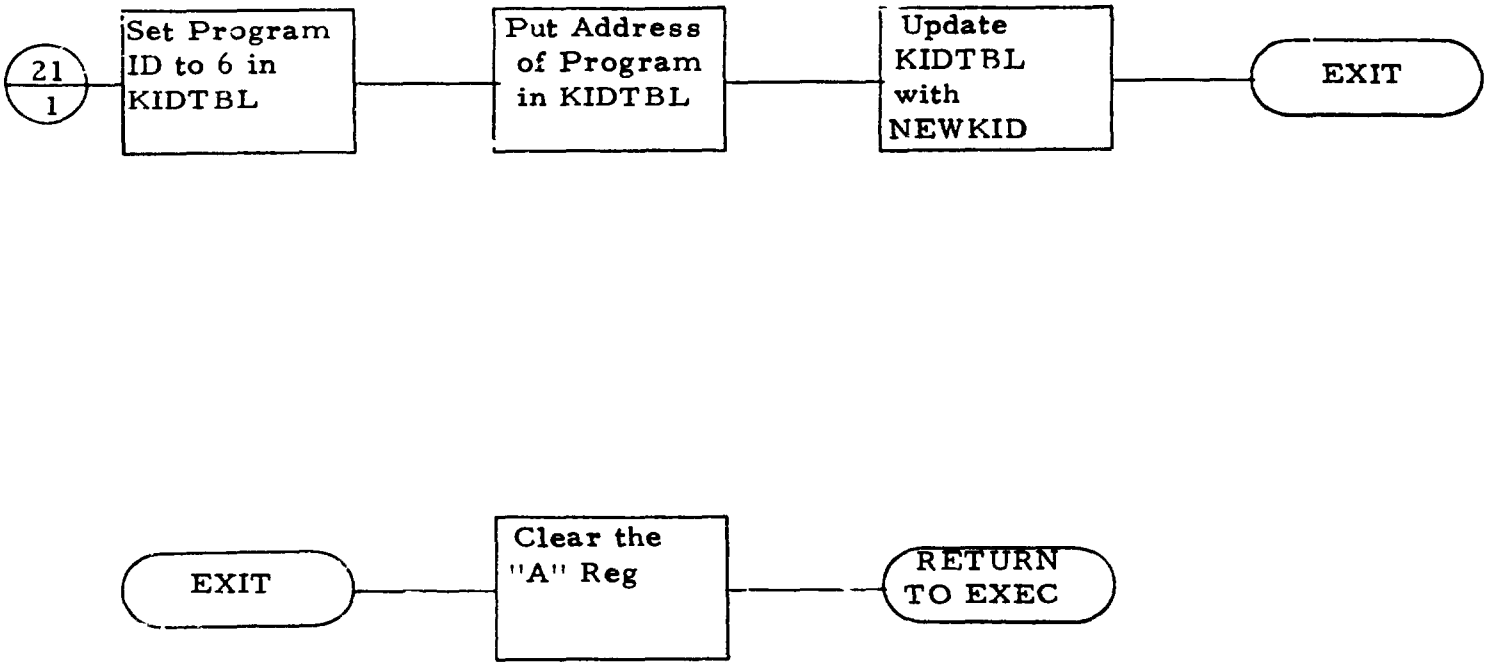


Figure 7-4 (Cont'd)

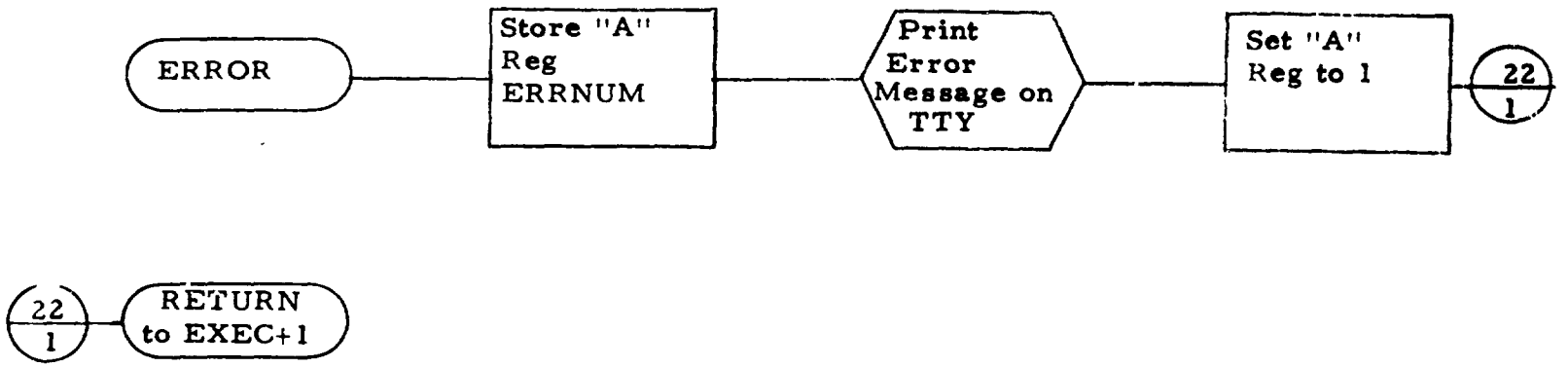
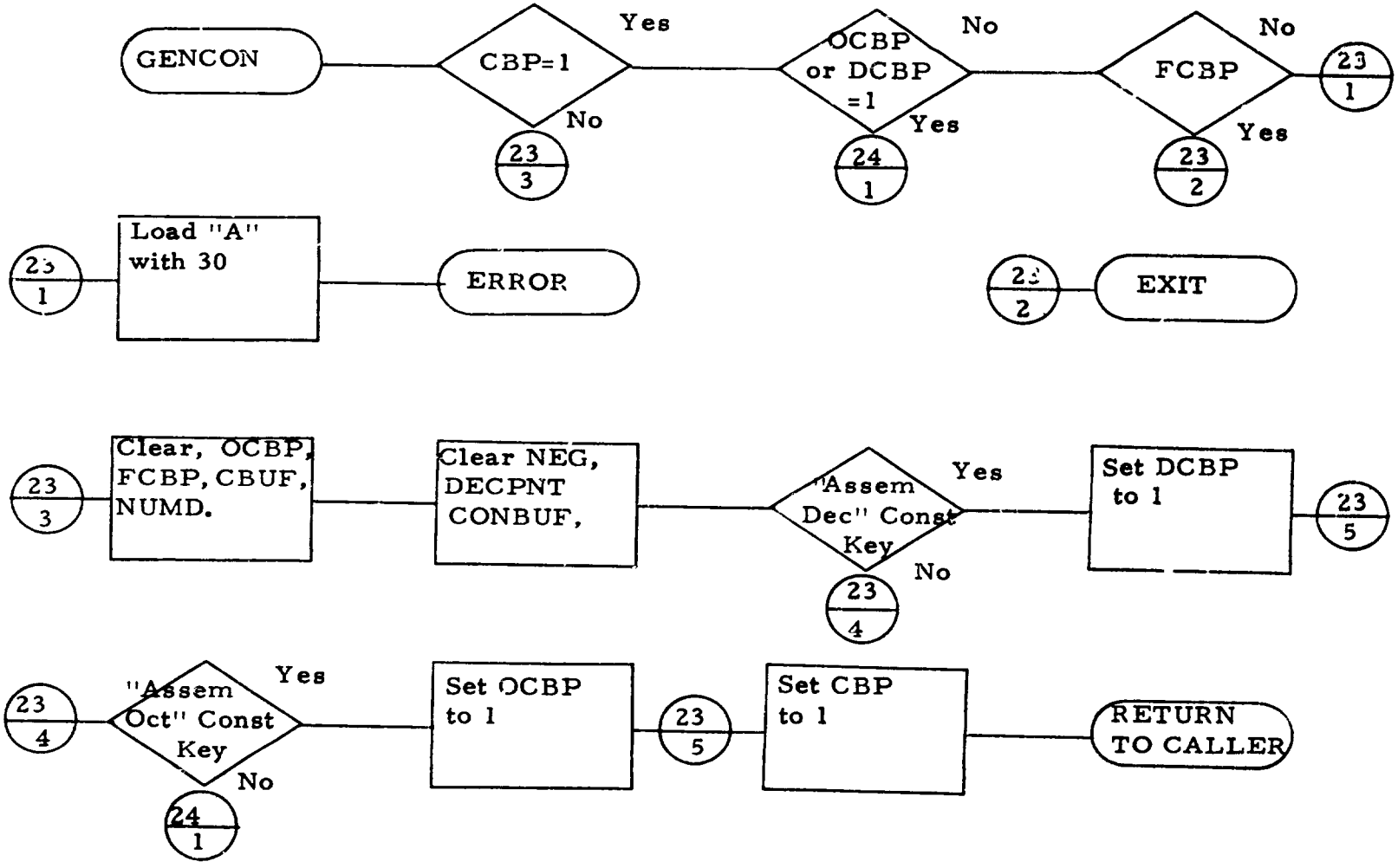


Figure 7-4 (Cont'd)



Constant Generation
Figure 7-5

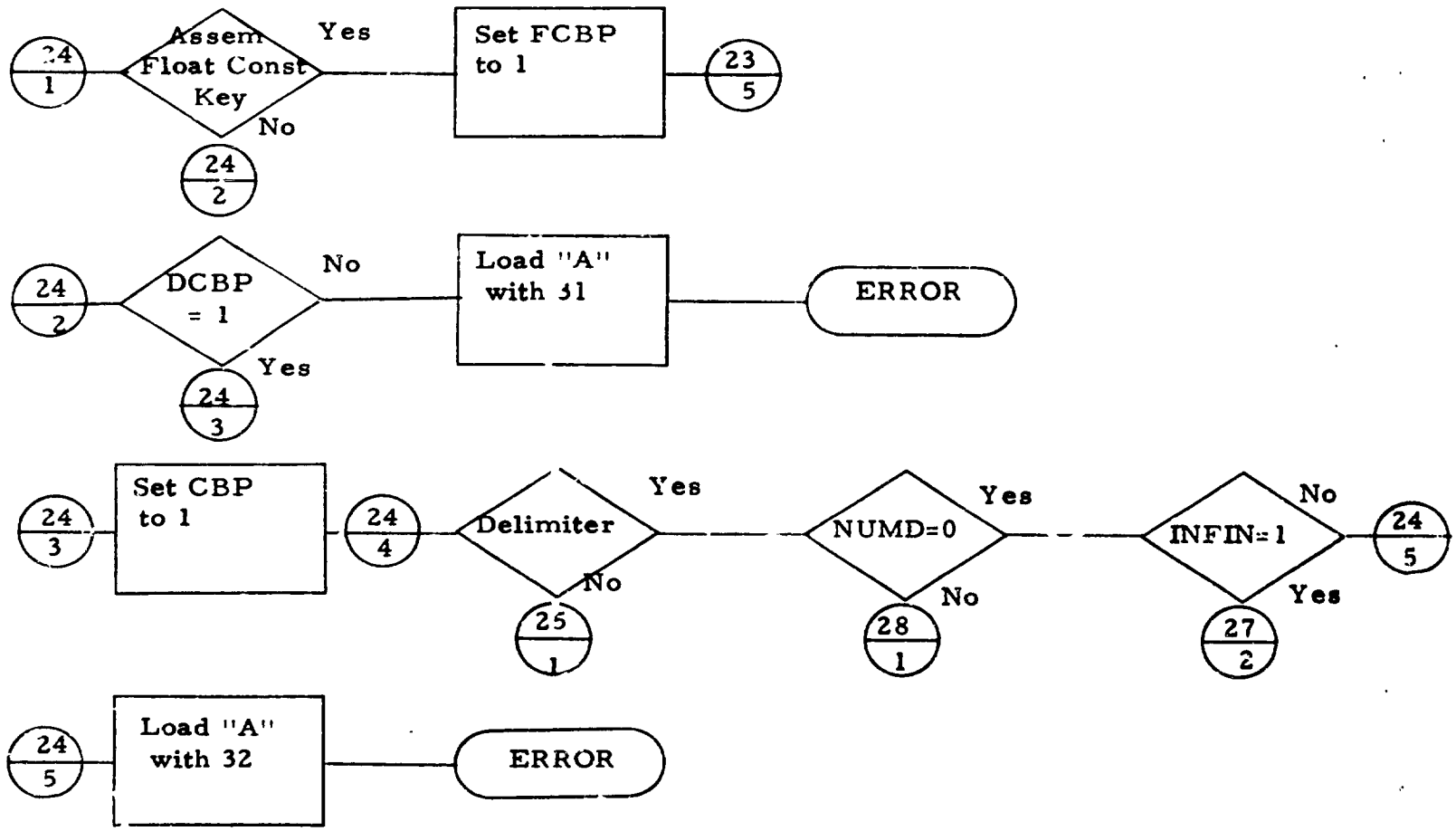


Figure 7-5 (Cont'd)

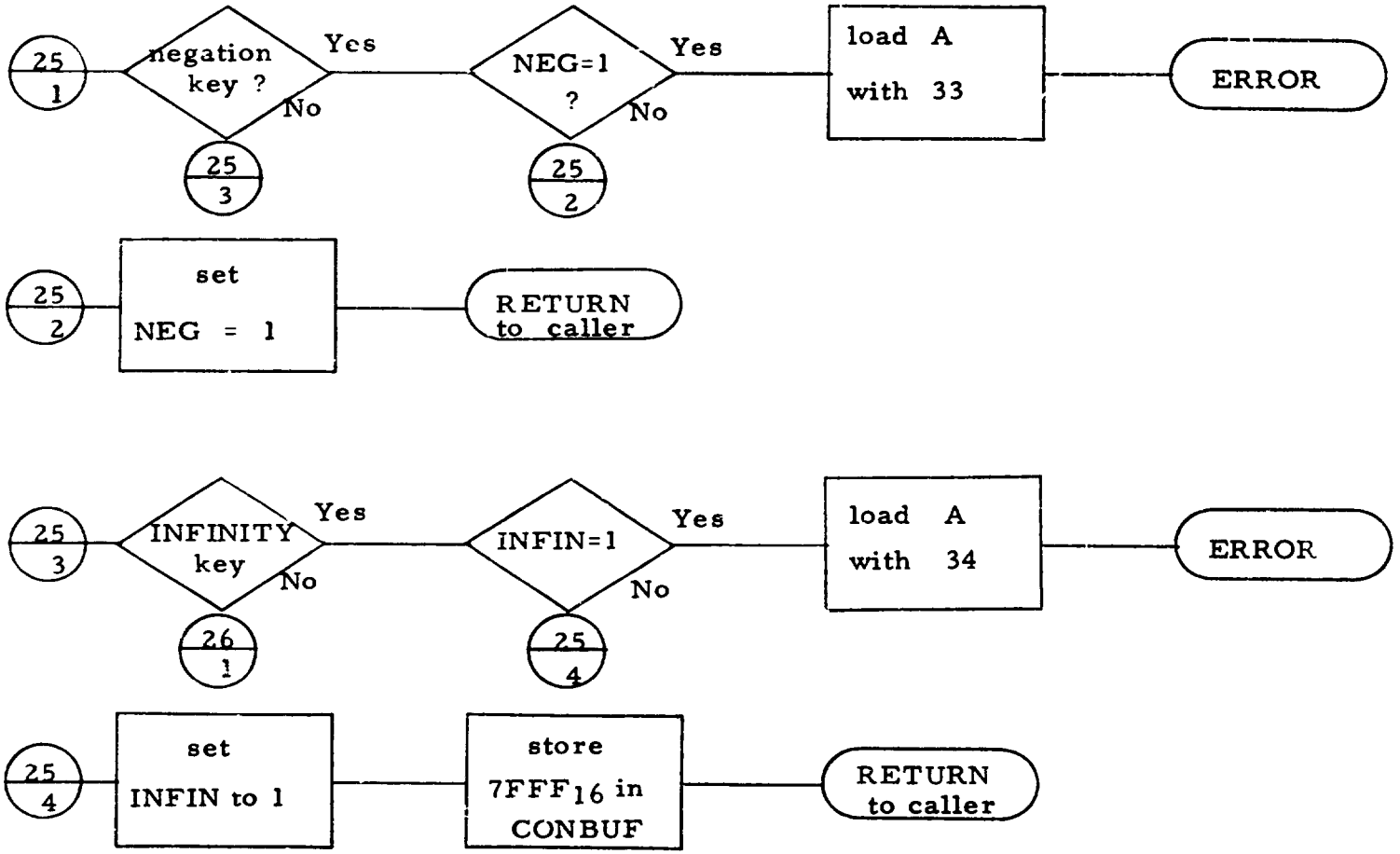


Figure 7-5 (Cont'd)

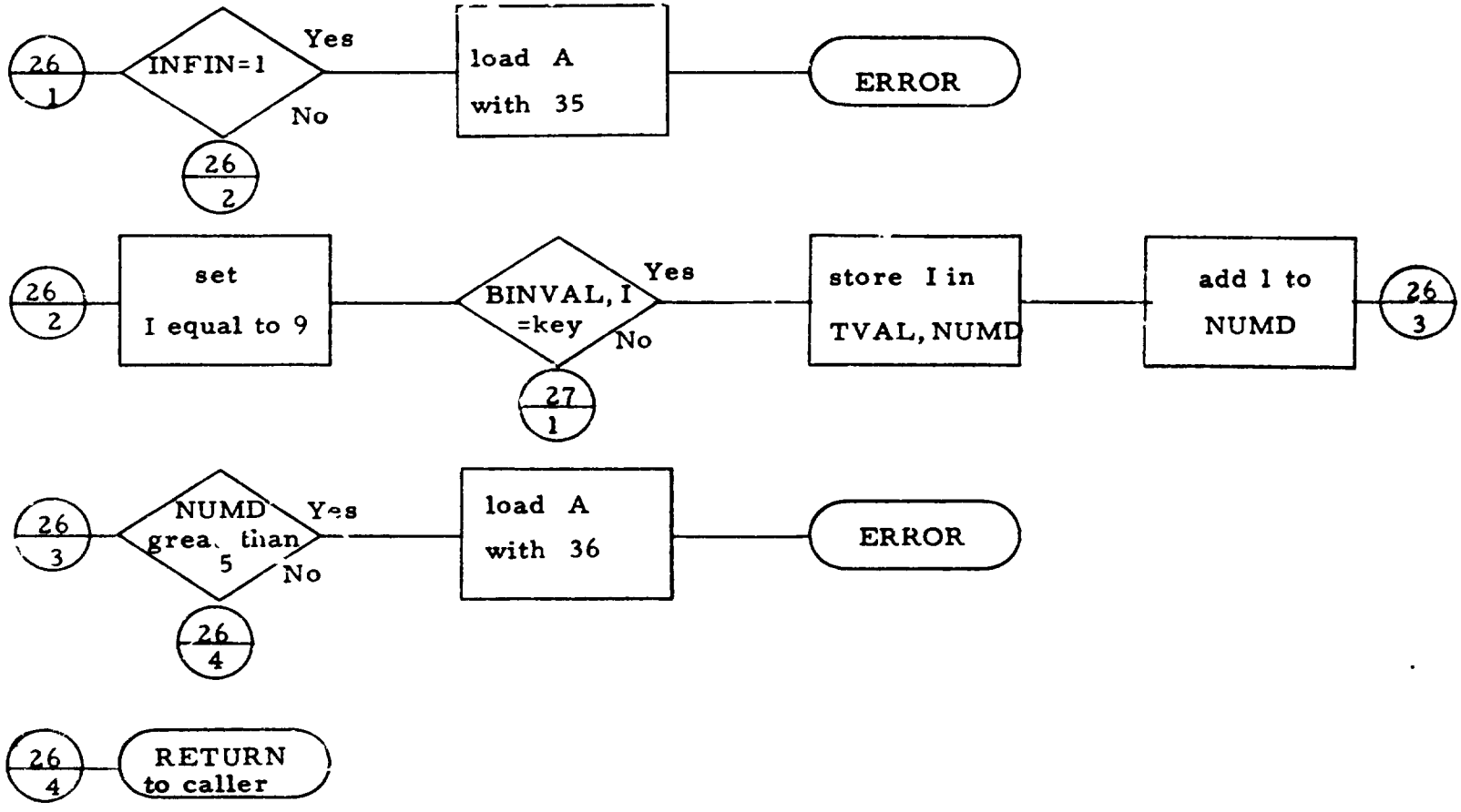


Figure 7-5 (Cont'd)

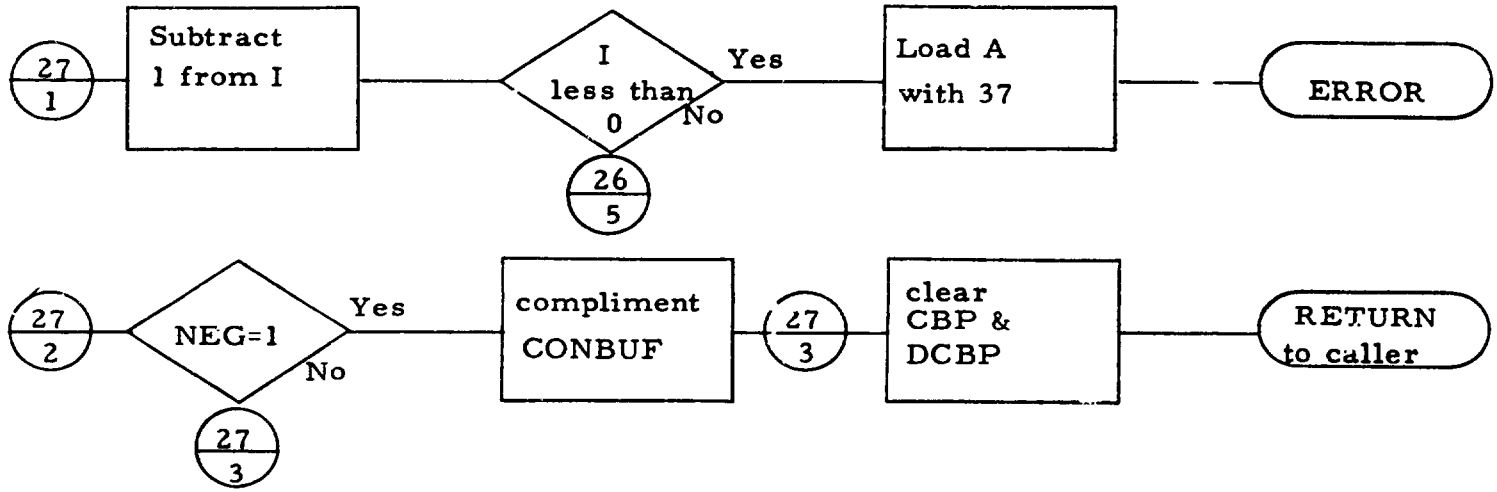


Figure 7-5 (Cont'd)

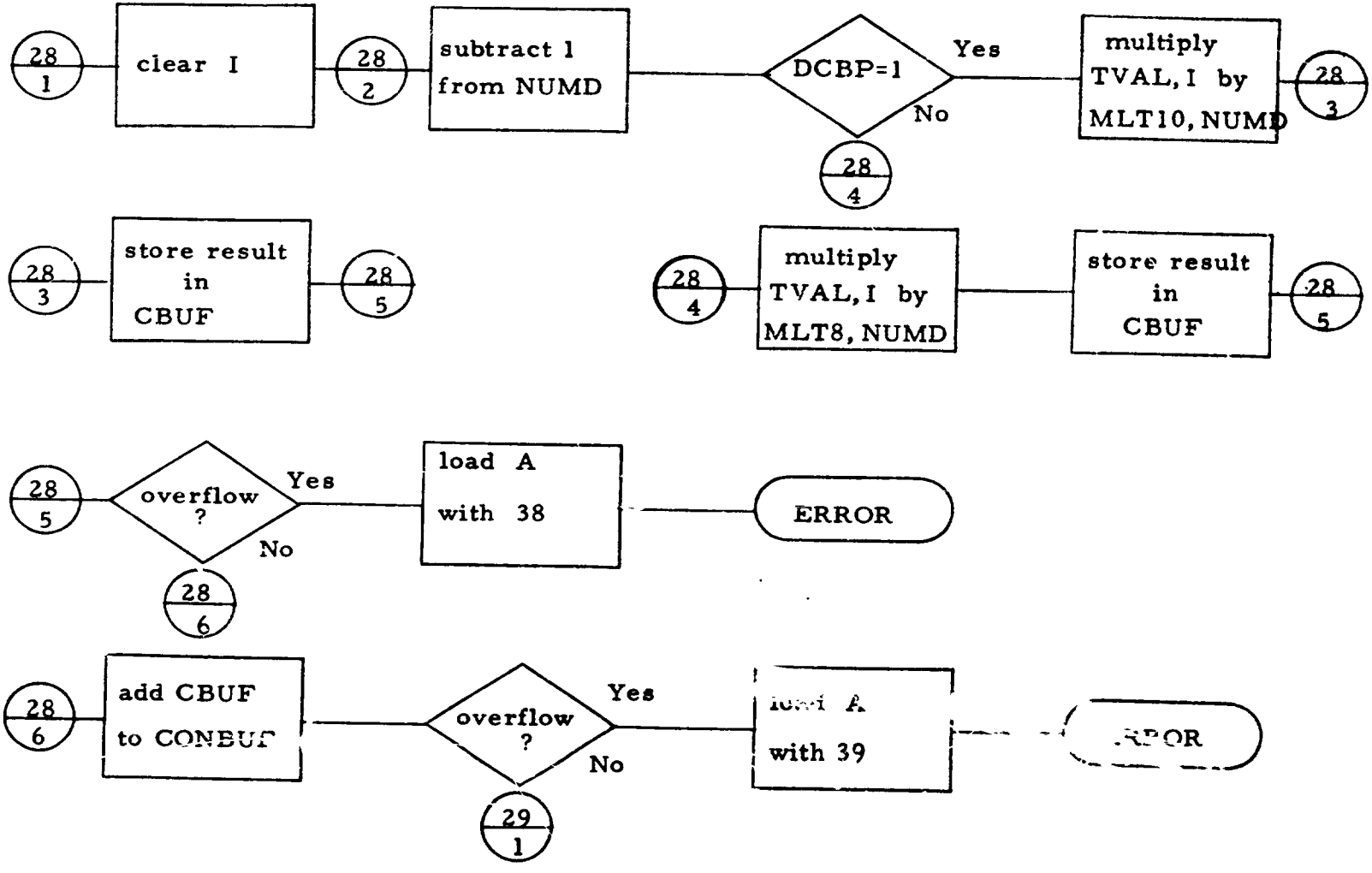


Figure 7-5 (Cont'd)

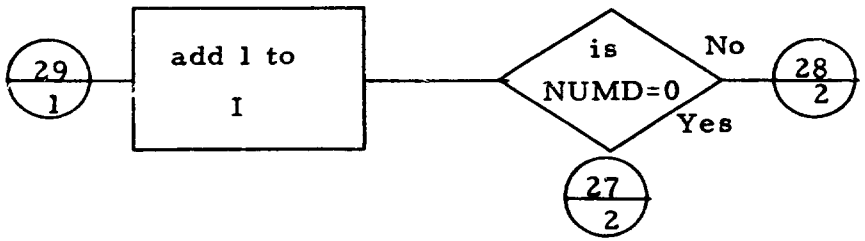


Figure 7-5 (Cont'd)

SECTION 8

IDL LOADER

8.1 GENERAL

The purpose of the Loader routine is to Load into user core an IDL application program. The application program is stored in the disk file in IDL loadable format (Section 5). The Loader is responsible for the assignment and allocation of all user core. It must also update the key identification table to reflect the assignments made. When an error free load is completed, the Loader notifies the Executive which may then begin execution of the application program.

This discussion will be divided into the six functions performed by the Loader. It should also be noted that if an error occurs during any phase of the loading process the job will be aborted. A complete set of flow charts for the detailed processes involved are available (Figure 8-1).

8.2 FUNCTIONS OF THE LOADER

These six functions of the Loader routine are as follows:

- o Program Lookup
- o Storage Assignment
- o Constant Assignment
- o Transfer Vector Loading
- o Command Loading
- o Transfer Vector Assignment

8.2.1 Program Lookup

The routine is entered with the key signature of the program to be loaded in KEY. KEY is used as an index into the key I/O table to obtain the disk address of the program.

Using this address information, the first two sectors of the program will be read into the buffer area called DSKBUF. The first 7 locations of DSKBUF will contain information about the make up of the remaining package. Section 5 defines this information. This information is now transferred to an area called DSKINF for future reference by the Loader routine.

8.2.2 Storage Assignment

Immediately after the Loader information will be the code ($FFFC_{16}$), which indicates the beginning of the storage packet. The next word will contain the number of words in the storage packet to be processed. Each key signature in the storage packet is read and a starting location of available core is assigned to its entry in the key identification table. A determination is made to see if the associated storage was preset. If preset storage is indicated, the preset data is transferred from DSKBUF to program core starting at the location assigned to that key. The location counter is updated to reflect each transfer. If storage is not preset, the location counter is increased by the amount of the storage assigned to that key. This process continues until all storage has been assigned. If the storage packet exceeds the first two disk sectors, the next consecutive two sectors will be read from the disk and the processing continued.

8.2.3 Constant Assignment

Immediately following the last entry of the storage packet there will be the code ($FFFF_{16}$) for the constant packet. Its contents will have been previously declared by the user writing the program. The handling of constants is straight-forward. The starting core location of the constant block is set in a register called CONLOC where it may be referenced later by the Interpreter. Then each word of the block is transferred from DSKBUF to program core. If the constants

are so numerous as to exceed the DSKBUF, the next two consecutive sectors will be read from the disk into DSKBUF and the process repeated.

8.2.4 Transfer Vector Loading

Sequentially the next packet to be processed is the transfer vector packet. At this time not much is done with it except to load it directly as is into program core. The first location address of the packet is set to a core location called JMPLOC. Nothing will be done with the assignments until the Commands have been retrieved from the disk and relocated in core.

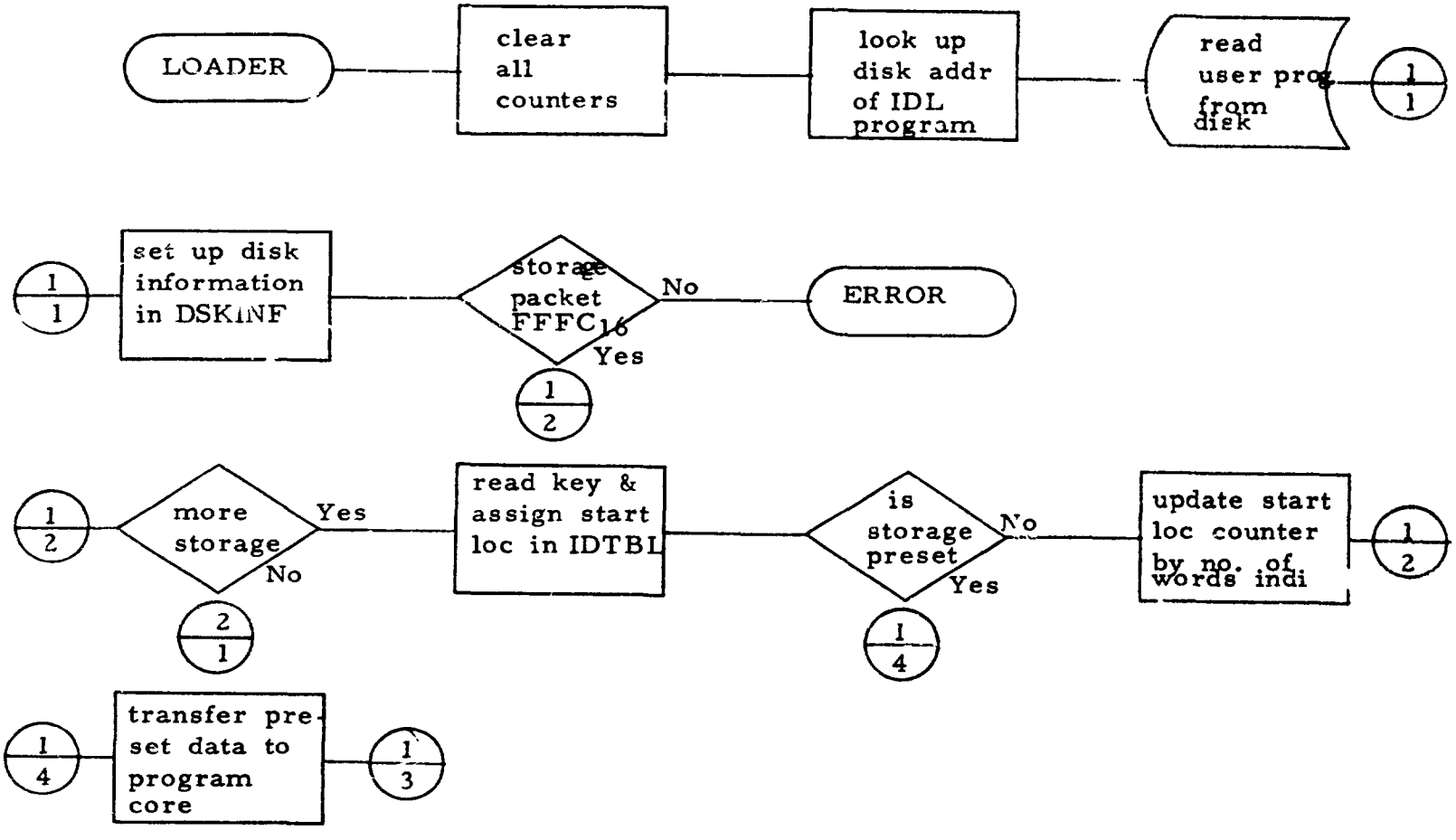
8.2.5 Command Loading

Upon completion of the transfer vector loading, the commands will be loaded. The sector address is available from DSKINF along with the number of words in the packet. The entire packet will be read directly to program core at one time. No transfers from one buffer to the other or any processing is necessary. With the commands in, the users' complete program is resident in core. Now it is possible to make transfer vector assignments.

8.2.6 Transfer Vector Assignment

Each program may contain other programs. The programs contained in any program may be made up of any number of previously compiled programs, therefore, any number of jump packets may be encountered within a single program. After the entire program is in core, the Loader will go through the program Command by Command looking for a "Label Follows" Command. When a "Label Follows" Command is encountered the jump table is searched for a matching label. When a match is found, the address of the Command following the "Label Follows" Command is placed in the appropriate location of the jump packet. If a new jump packet is encountered the pointer to

the last jump packet is stored away and the labels of the new packet are assigned. This will hold true for all new jump packets encountered. When end of subroutine is encountered the address of the previous jump packet pointer is read up and the current one discarded. The jump packet pointers are stored in DSKBUF while the current table address is in ACTJMP. When the end of the main program is encountered control will be given to the Executive.



IDL LOADER
Figure 8-1

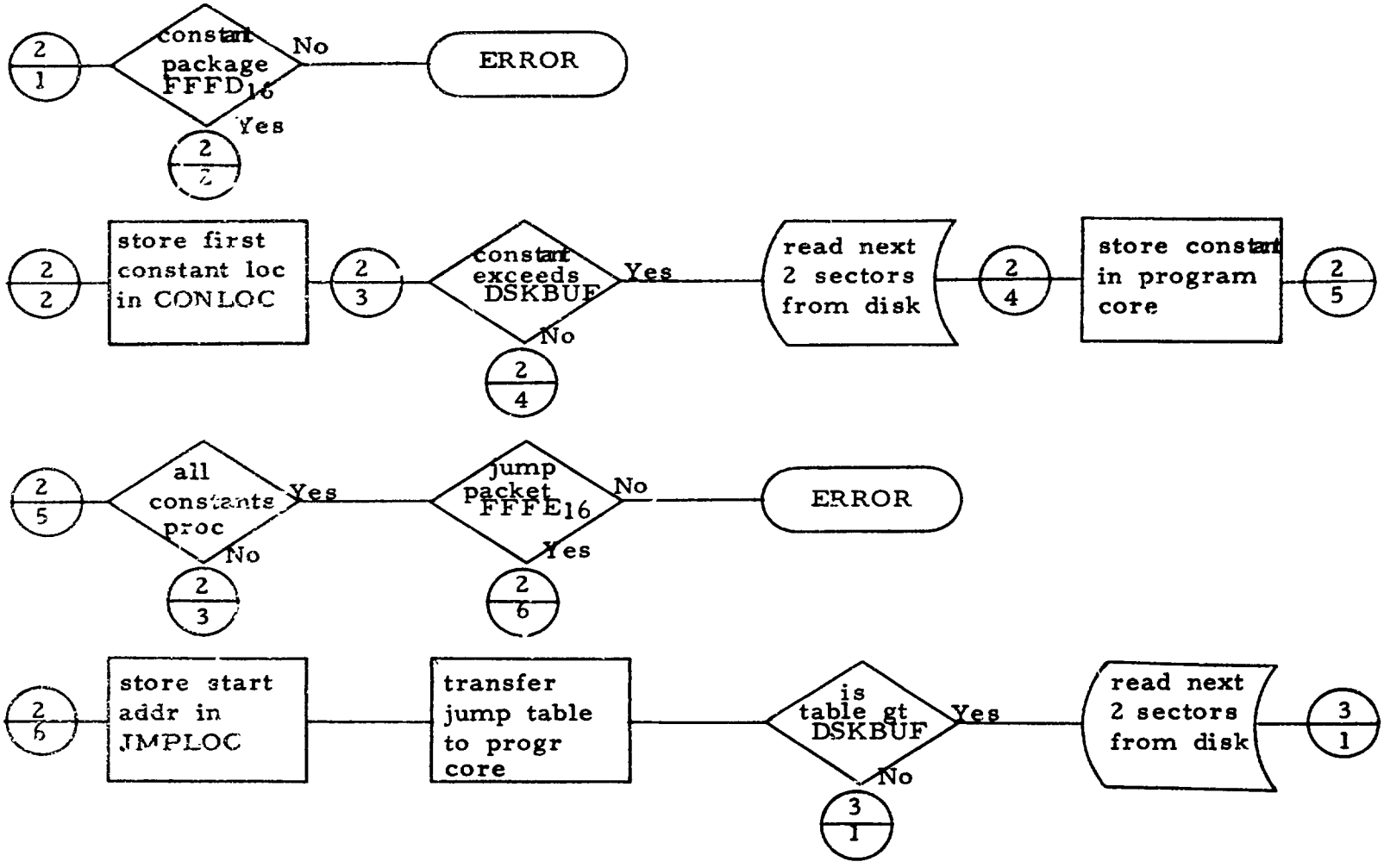


Figure 8-1 (Cont'd)

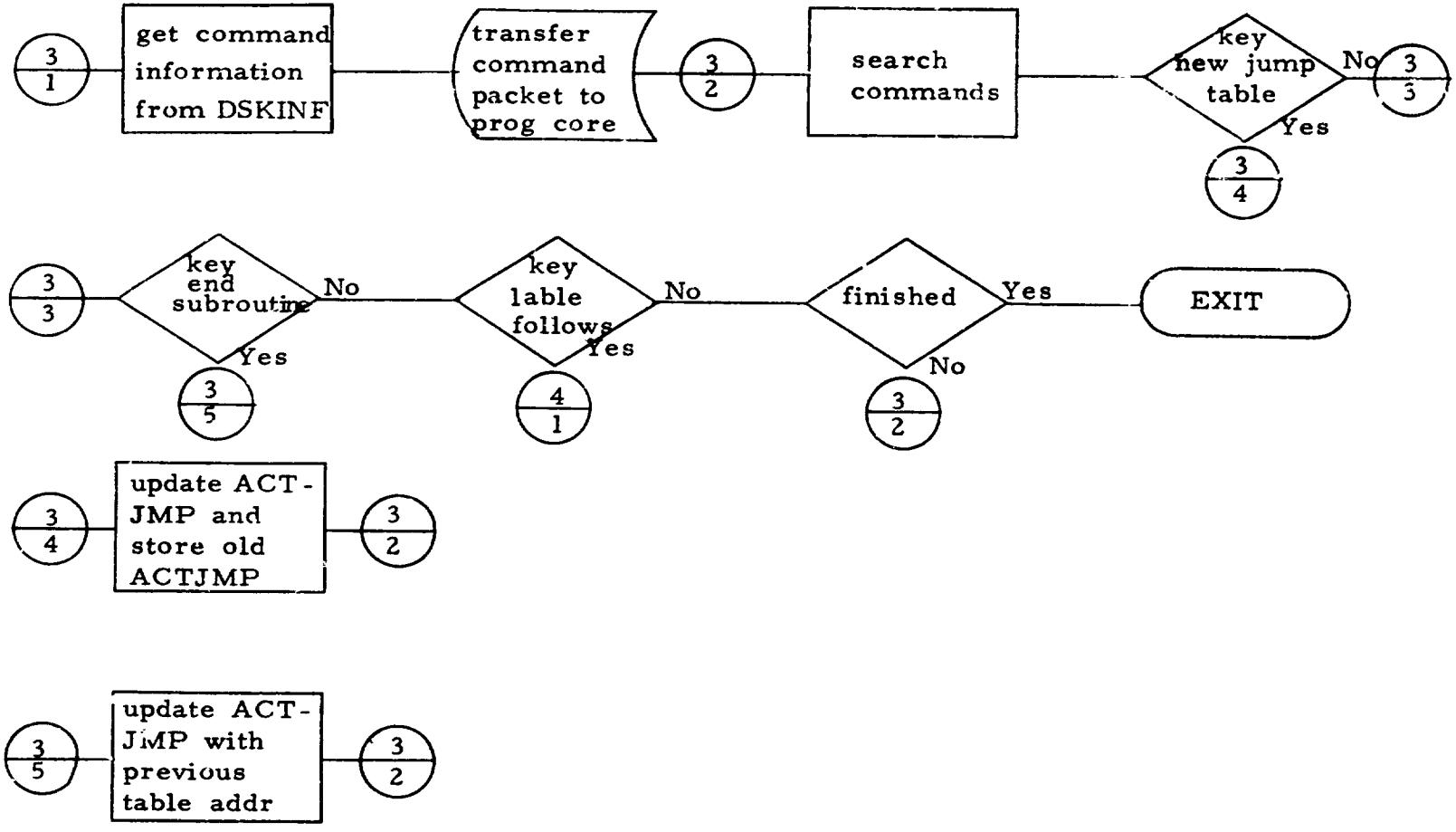


Figure 8-1 (Cont'd)

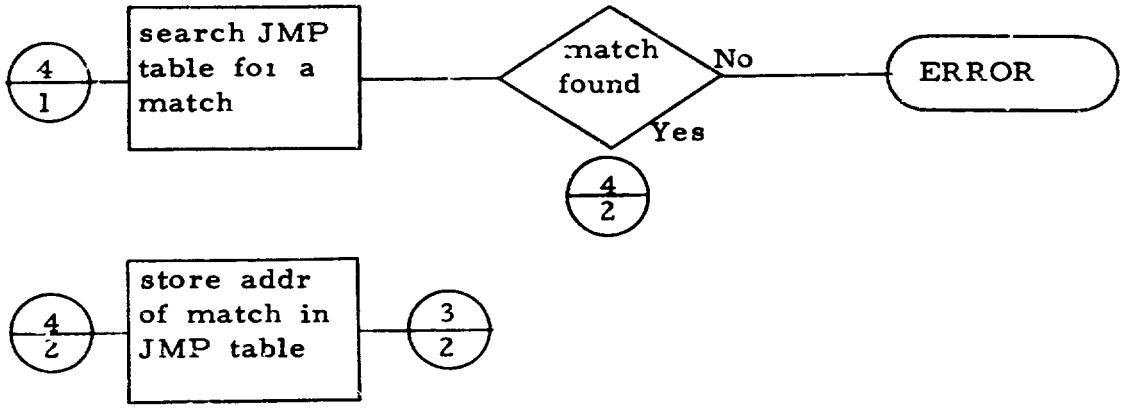


Figure 8-1 (Cont'd)

SECTION 9

IDL INTERPRETER

9.1 GENERAL

The Interpreter along with the Commands make up the psuedo computer which will execute the user's application program. It is the function of the Interpreter to read the Commands, form an effective address from the parameters and jump to the appropriate Command Routine. The Interpreter is also responsible for processing some special IDL keys. Keys which the Interpreter handles alter the sequence of the program flow. These keys are "Break Point", "Pause", and all the jump keys.

9.2 PROGRAM SEQUENCE

In order to keep track of the program flow, a register is set to the location of the first Command in the Command packet. This address has been set by the Loader in SRTLOC. It is now transferred to PCNT. PCNT is incremented each time a Command is processed. Each new key to be processed will be loaded from the address contained in PCNT. After the successful processing of a Command key (type 2) the syntax length for that key will be added to PCNT, resulting in the address of the next Command. In the case of a key which has no syntax 2 will be added PCNT.

Several keys have no syntax, but they occupy two words in the Command packet.

Subprograms may be imbedded within the Command packet. If a subprogram is found, it is necessary o reflect this in PCNT as well as other registers. The constant packet and Transfer packet will be imbedded within the Command packet being processed.

The length of these packets is added to PCNT. Their respective absolute core locations will be placed in the active packet locators, (ACTCON and ACTJMP). This operation is necessary because IDL jumps can only be made to locations within a subroutine. The target address of the jumps are carried in the transfer packet. Program flow proceeds as indicated earlier until the end of subroutine is found. At this time, the contents of ACTCON and ACTJMP are set back to the previous addresses. Any number of subroutines may be nested in IDL, and the Interpreter will keep track of the location of all of them.

9.3 SPECIAL CASES

To process a jump, the Interpreter searches the Transfer packet for target of the jump. This target address is stored in PCNT and processing continues from there. "Pause" and "Break Point" both cause the program flow to halt. Because the conditional switches available are not on the CDC 1700 all "Break Point" keys stop the flow regardless of the conditions. To resume, the start switch is set and a count of two is added to PCNT.

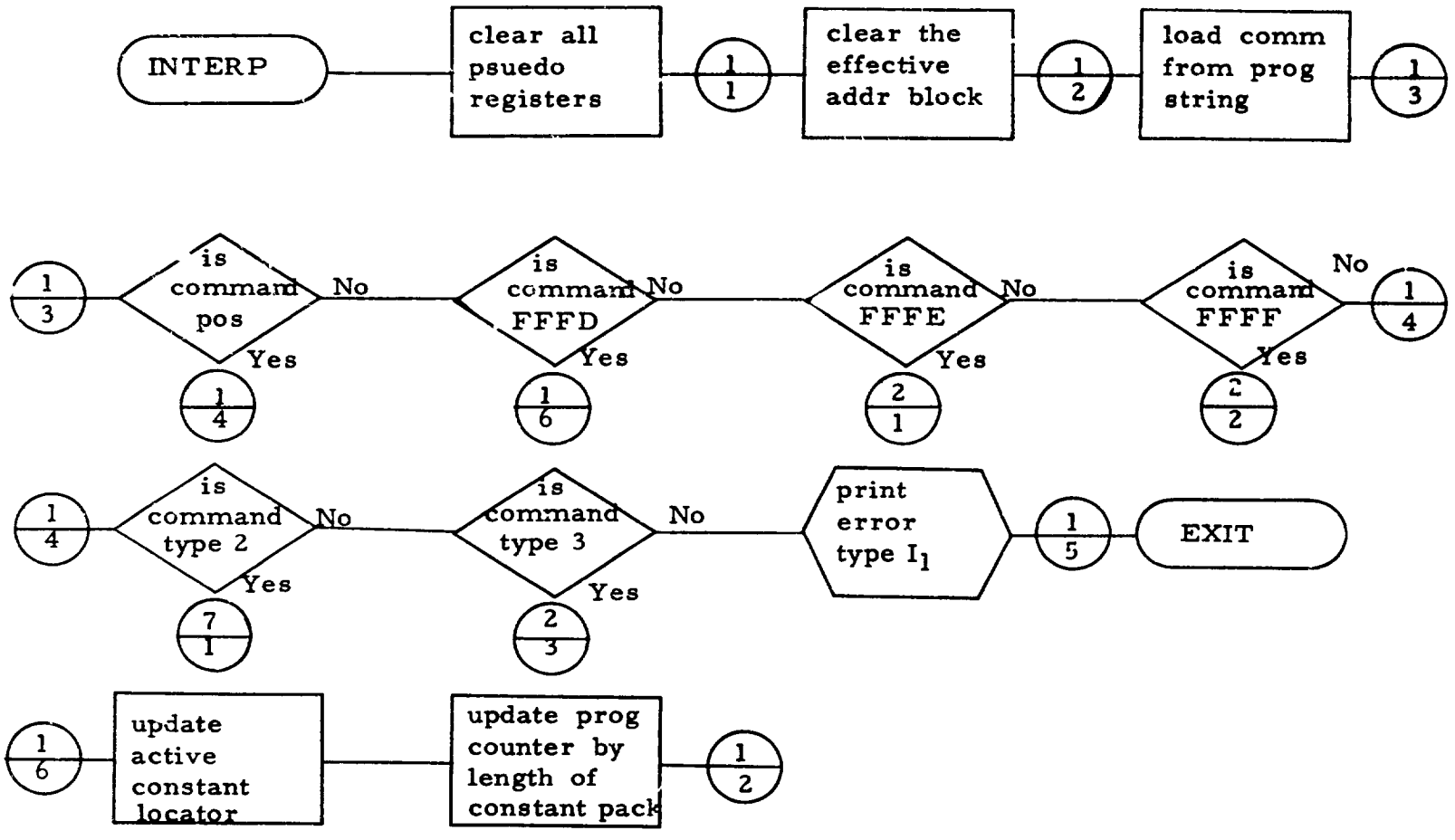
9.4 COMMAND CALLS

The IDL Commands make up the bulk of the instructions to be handled by the Interpreters. The general procedure is to set up an effective address from the key parameters then do a return jump to the indicated Command Routine.

The parameters are read from the Command packet. A determination is made as to their type, that is, register, block, constant or zero. At this point, each type is handled slightly different. If the parameter is a zero the address of a zero, ZROADR, is stored in the effective address block (EFADBK). If the parameter is a constant its absolute address is obtained from the constant packet.

The address of the constant is then stored in the EFADBK. When the parameter is a register or block, the signature portion of the parameter is used as an index into the Key ID Table. The absolute starting address of that block or register is obtained from the Key Identification table. Once the absolute address is obtained it is stored in the EFADBK.

After all the parameters have been processed a return jump to the Command Routine is performed. The Command Routine forms the effective address from the entries of EFADBK and performs its action upon it.



IDL Interpreter
Figure 9-1

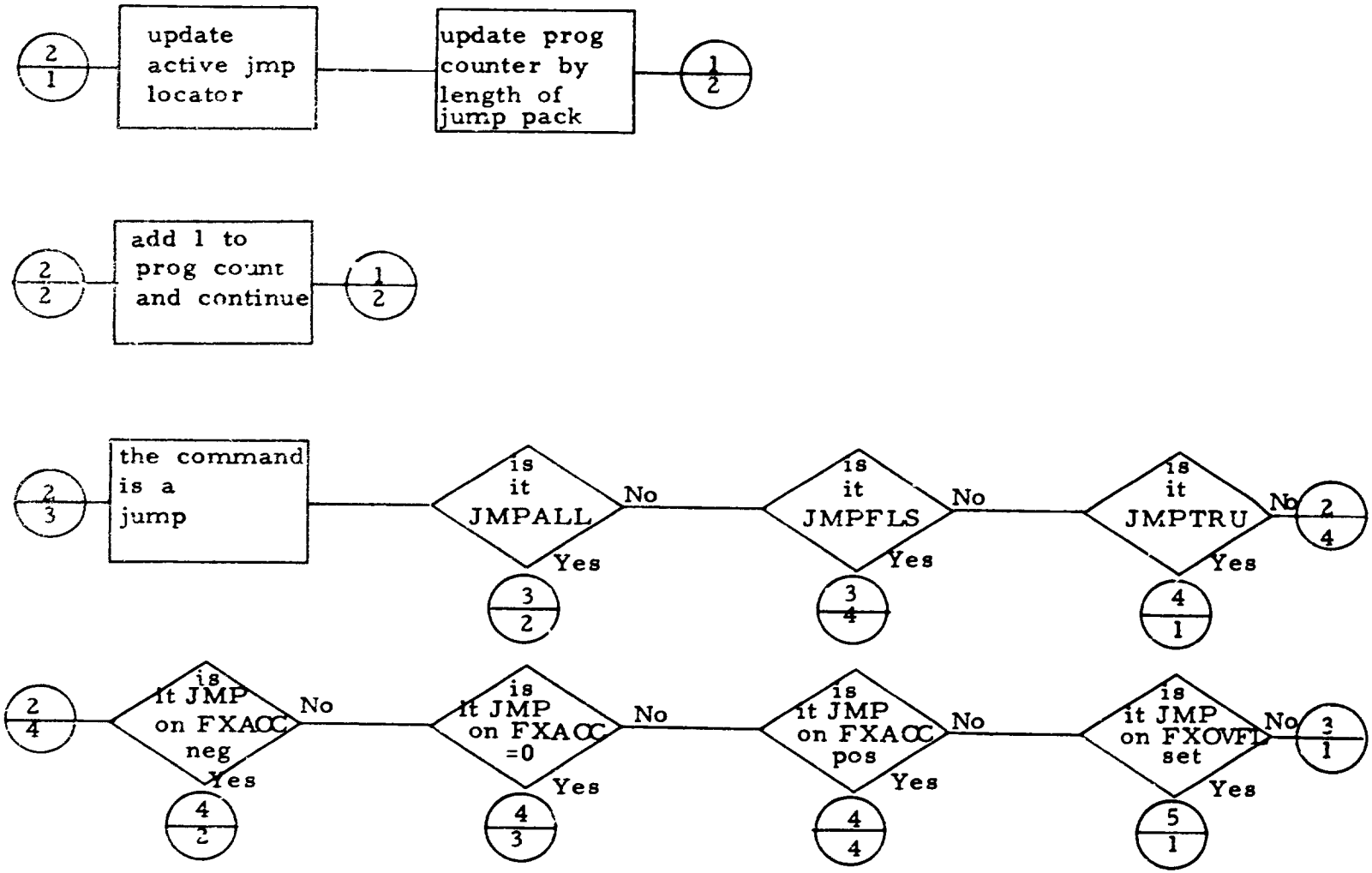


Figure 9-1 (Cont'd)

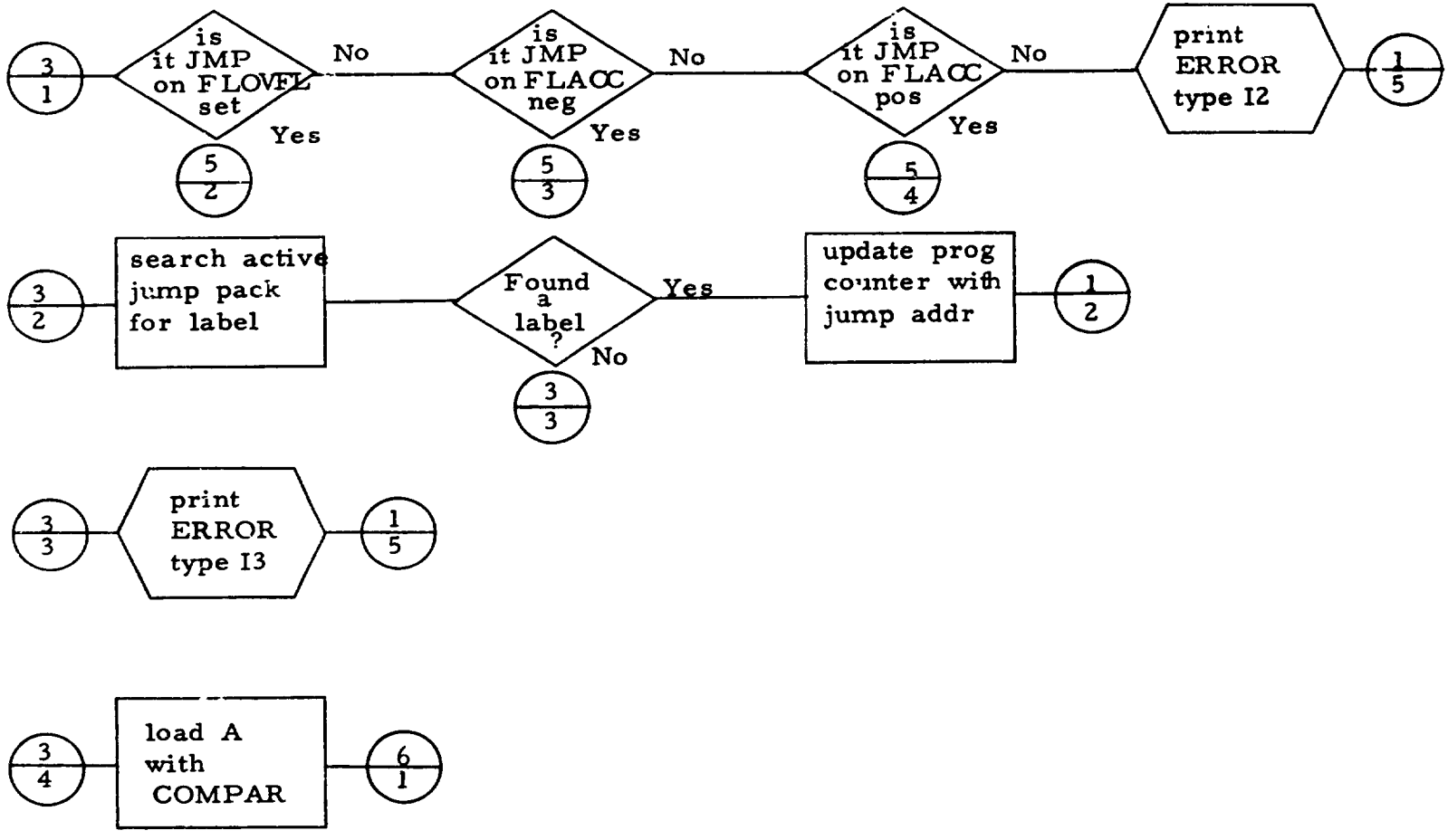
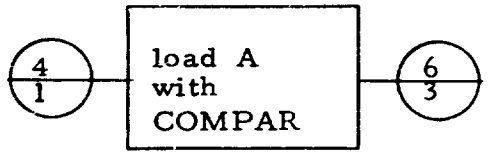
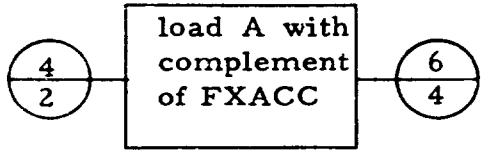
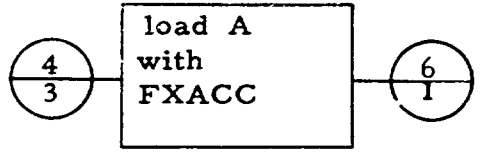


Figure 9-1 (Cont'd)







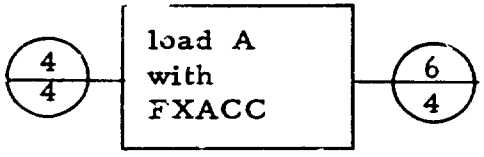
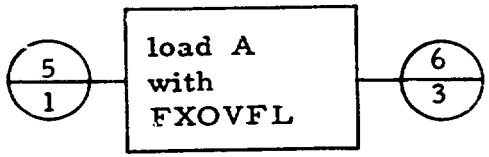
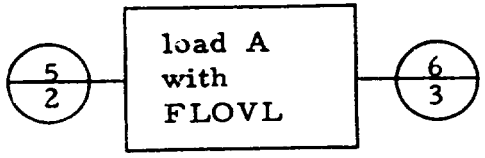
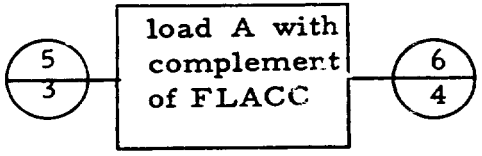


Figure 9-1 (Cont'd)







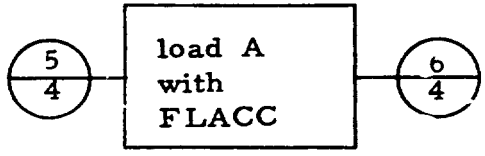


Figure 9-1 (Cont'd)

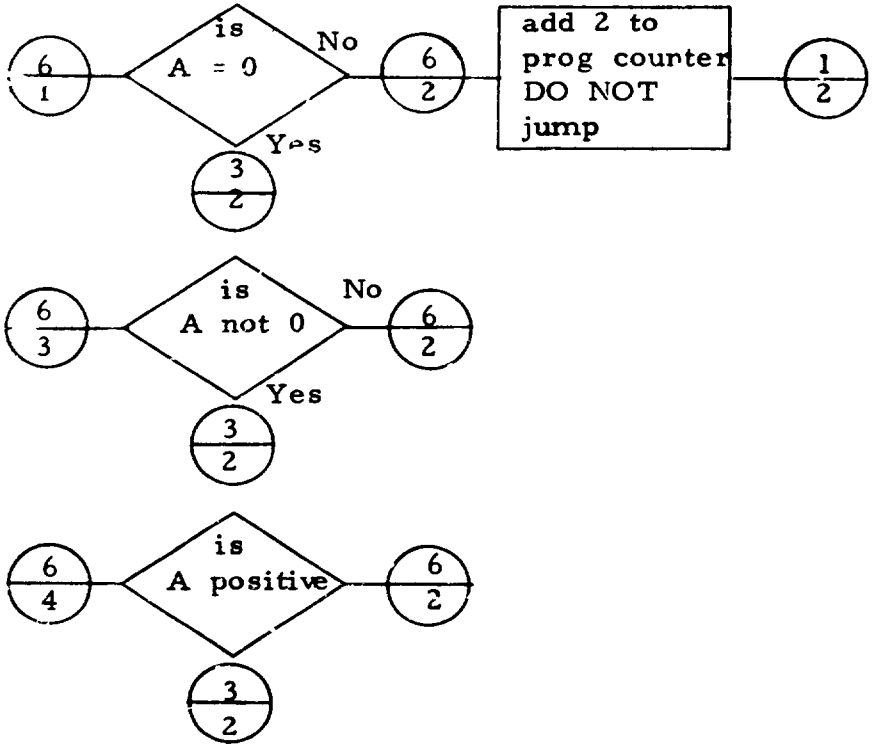


Figure 9-1 (Cont'd)

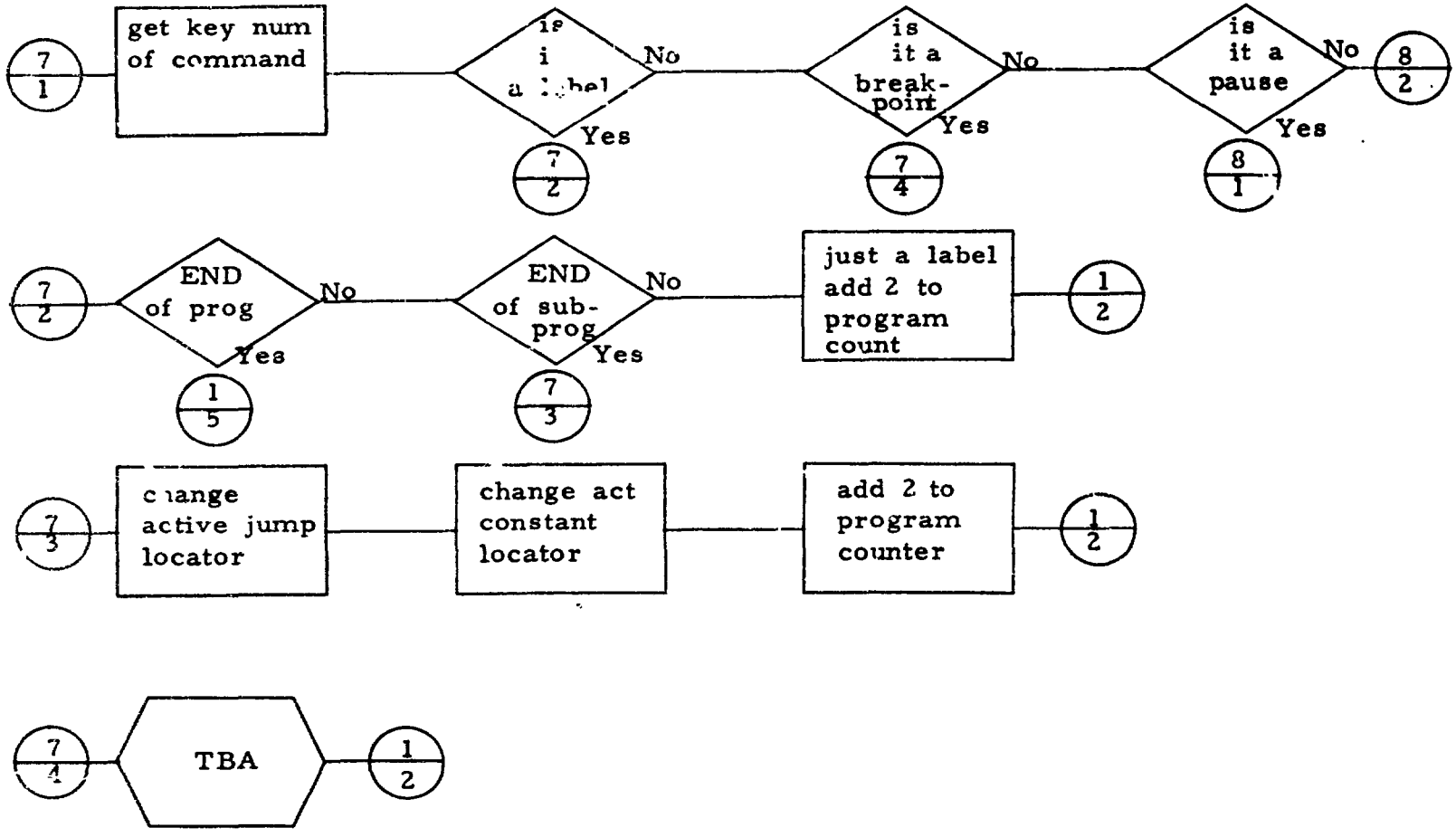


Figure 9-1 (Cont'd)

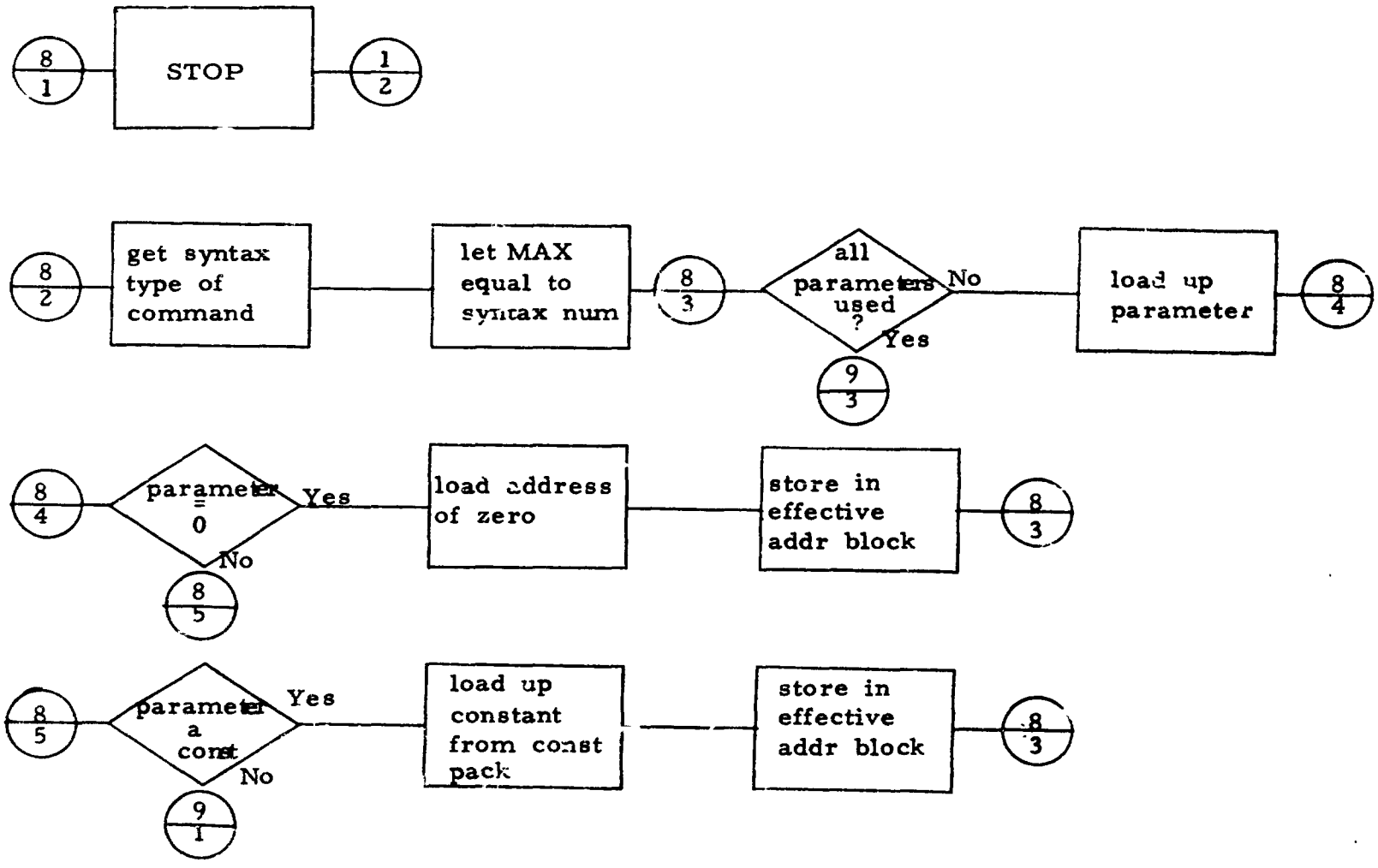


Figure 9-1 (Cont'd)

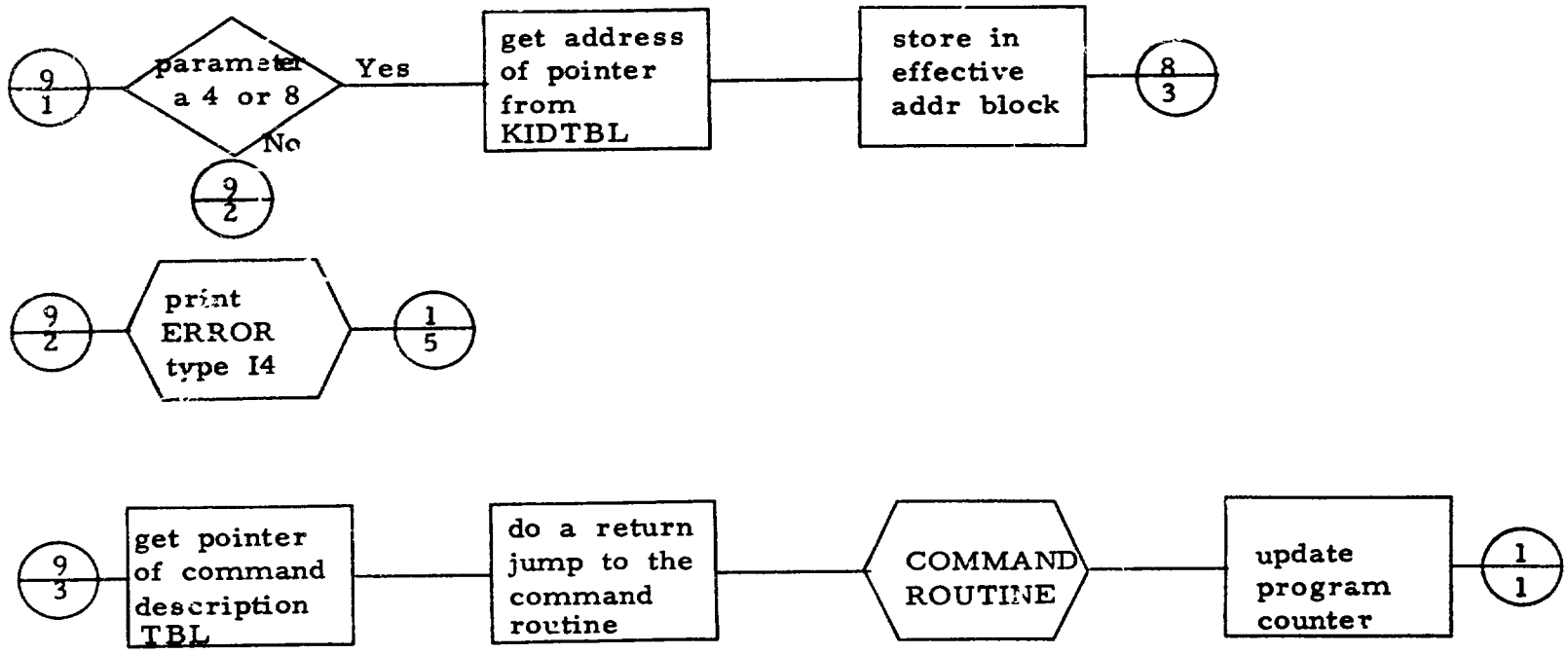


Figure 9-1 (Cont'd)

SECTION 10

IDL COMMAND ROUTINES

10.1 GENERAL

This Section fully describes specific IDL Command Routines discussed in Section 2.2.5. Many of the routines have been grouped into categories. One example of this is the floating point arithmetic operations. They are described under Section 10.2.

Section 10.2 to 10.23 conform in format. This conformity lends itself to easier reading and understanding of the routine discussion. The format of the discussion is as follows:

- Program Name(s): The actual rotuine name and short description.
- Abstract: A general description of the routines' function.
- Operating Requirements: Computer System that the routines are implemented on.
- Language: Computer System language that routines are implemented in.
- Functional Flow Chart(s): Full descriptive flow charts of the routine.
- Program Description: A detailed prose description of the routine.

- o Inputs[1]: Data or Parameters needed in order for the routine to function.
- o Outputs: Description of resulting data of the routine.
- o Call Sequence[2]: Description of calling sequence of the routine including any parameters to be passed.

The reader is asked to note that all computer location labels are in capital letters, i.e. FXACC - the fixed point accumulator, etc. Whenever a buffer or block of data is addressed by its index, the flow charts will show same as "buffer, index" or "block, index".

---

[1] The user need not concern himself with this discussion since the IDL System is the only originator of the call to the command routines. Section 2.2.5 has discussed system call to these routines.

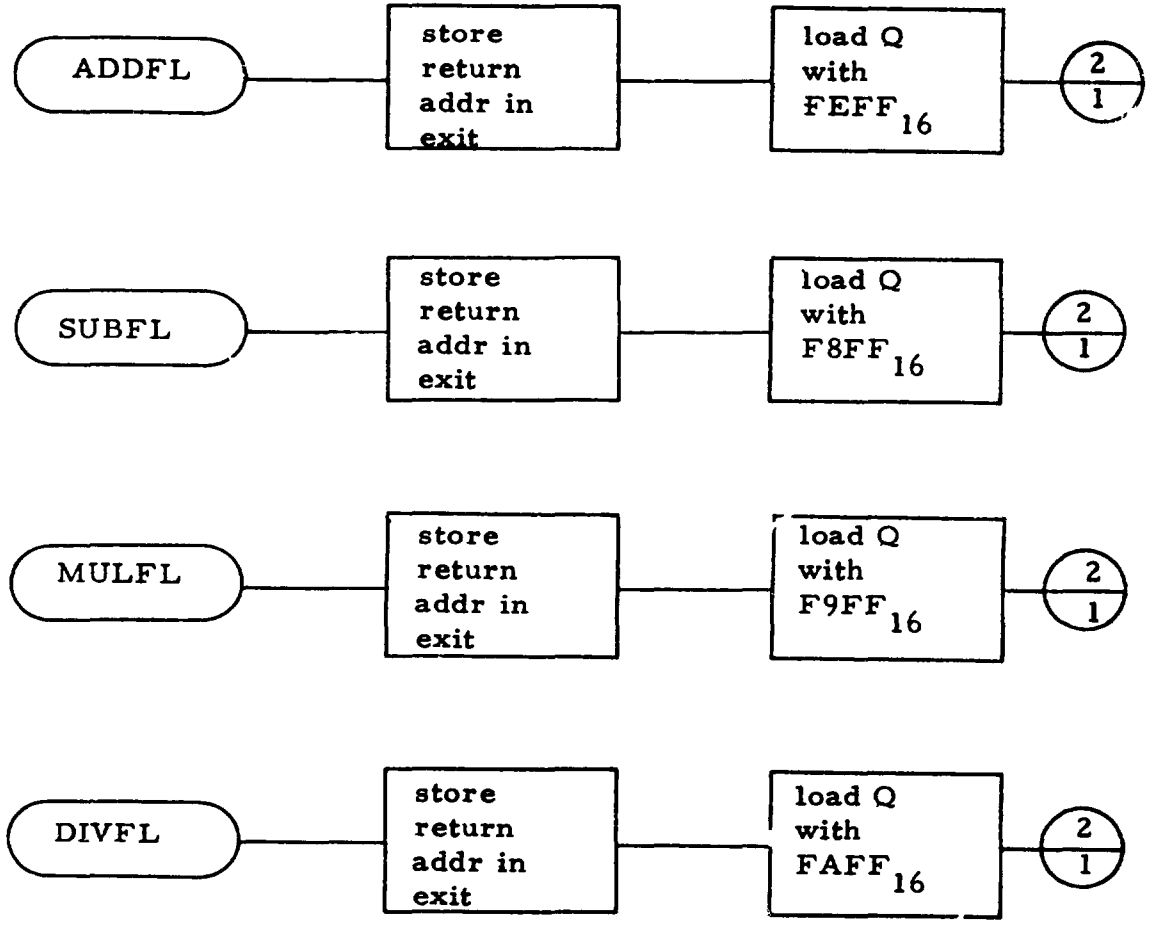
[2] All calls are in the form RTJ ROUTINE, where ROUTINE is the program name. IDLFLT and IDLUNF are the only programs that are called by other COMMAND ROUTINES. These need to have input data as described in the discussion of same.

10.2 FLOATING POINT ARITHMETIC

1. Program Names: ADDFL - Floating point add
   SUBFL - Floating point subtract
   MULFL - Floating point multiply
   DIVFL - Floating point divide
2. Abstract: The named programs perform their respective floating point arithmetic between the FLACC and the given operand. In case of overflow the overflow indicator is set.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-1
6. Program Description: Each program has its own entry point. A call to the CDC 1700 system program FLOT is made with the load-operation-store function code in call +1, and the address of FLACC stored in call +2 & 4. The address of the operand is stored in call +3. The result of the arithmetic operation is returned in FLACC. A check is made for overflow. If the operation resulted in overflow the floating point overflow indicator FLOVFL is set to 1. (In DIVFL, the operand is checked to see if it is 0. If so, FLOVFL is set to 1 and the operation is not performed). Exit.
7. Inputs: /BLOCK/INDEX/OPN/
8. Outputs: Result of the operation is stored in FLACC. FLOVFL is set to 1 on overflow.

9. **Call Sequence:**

```
RTJ     ADDFL or
RTJ     SUBFL or
RTJ     MULFL or
RTJ     DIVFL or
```



Floating Point Arithmetic

Figure 10-1

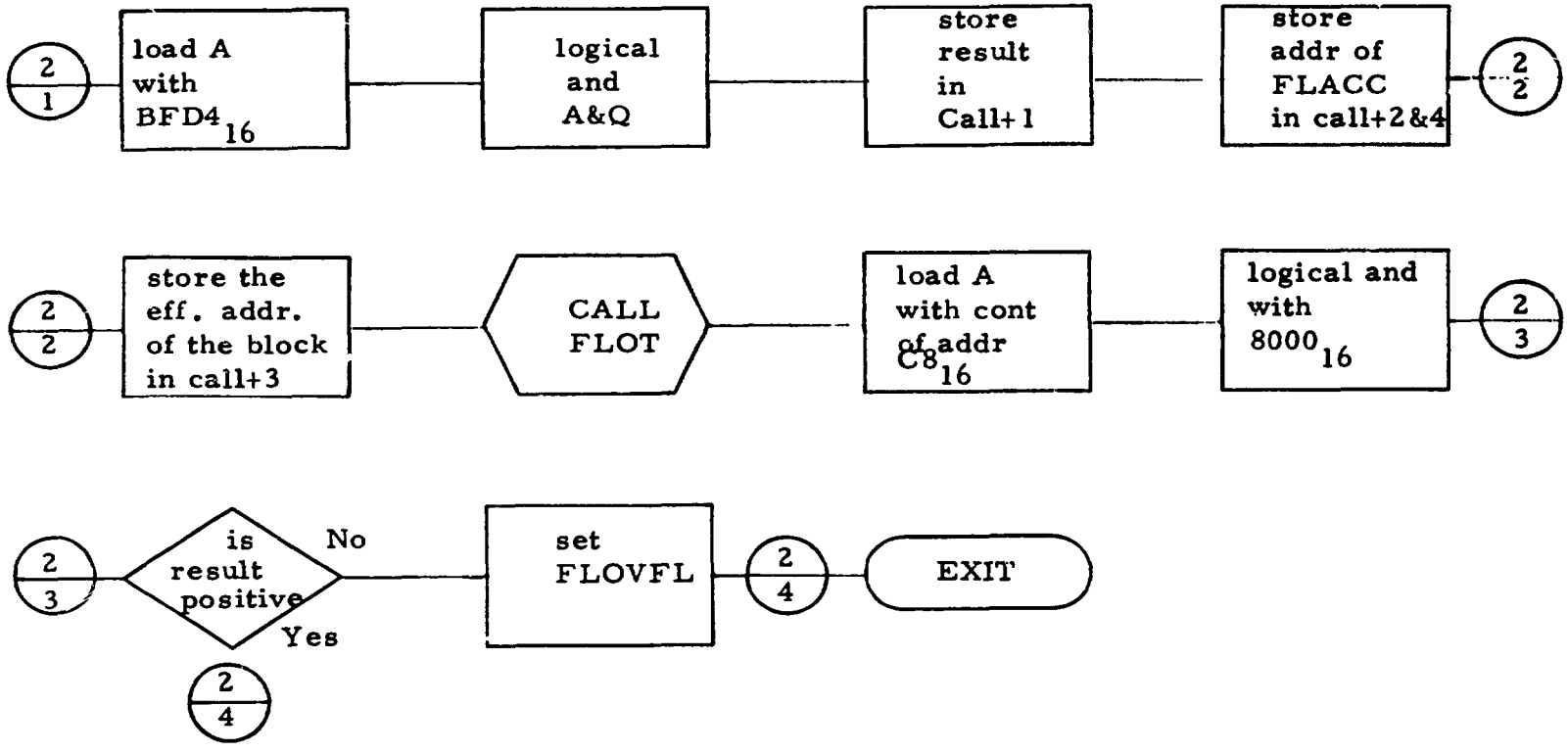
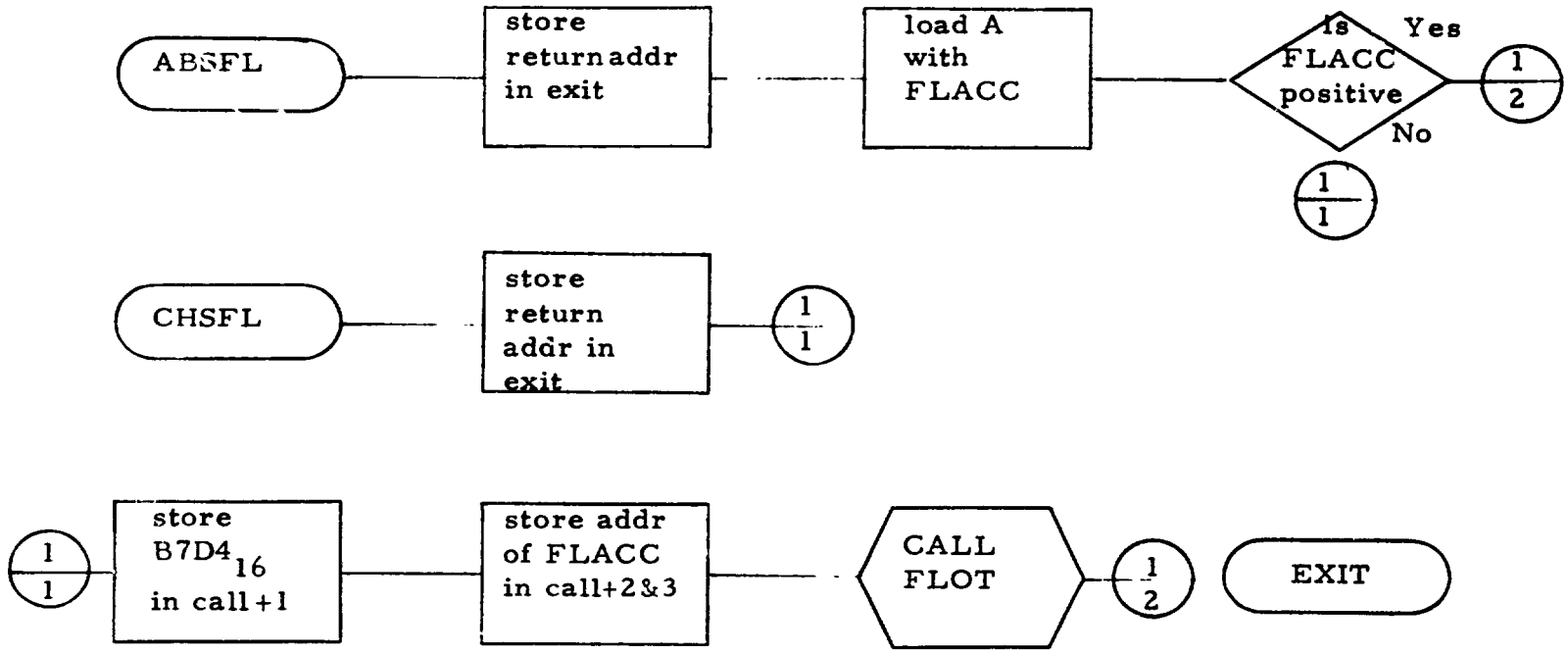


Figure 10-1 (Cont'd)

10.3 FLOATING POINT SIGN

1. Program Names: ABSFL - Absolute value of floating point number.
   CHSFL - Change sign of floating point number.
2. Abstract: The named programs perform their function on the floating point accumulator FLACC. ABSFL finds the absolute value of FLACC. CHSFL changes the sign of FLACC.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-2
6. Program Description: Each program has its own entry point. ABSFL checks the sign of FLACC. If it is positive it exits. If it is negative a call to the CDC 1700 system program FLOT is made with the load-complement-store code in call + 1 and the address of FLACC in call +2 & 3. The same call to FLOT is made for CHSFL. The result is stored in FLACC. Exit.
7. Inputs: /BLOCK/INDEX/OPN/
8. Outputs: Result of the operation is stored in FLACC
9. Call Sequence: RTJ ABSFL or
   RTJ CHSFL or



Floating Point Sign

Figure 10-2

10.4 FLOATING POINT COMPARE

1. Program Names: CFLG - Compare Floating point number if greater than operand.

   CFLL - Compare floating point number if less than operand.

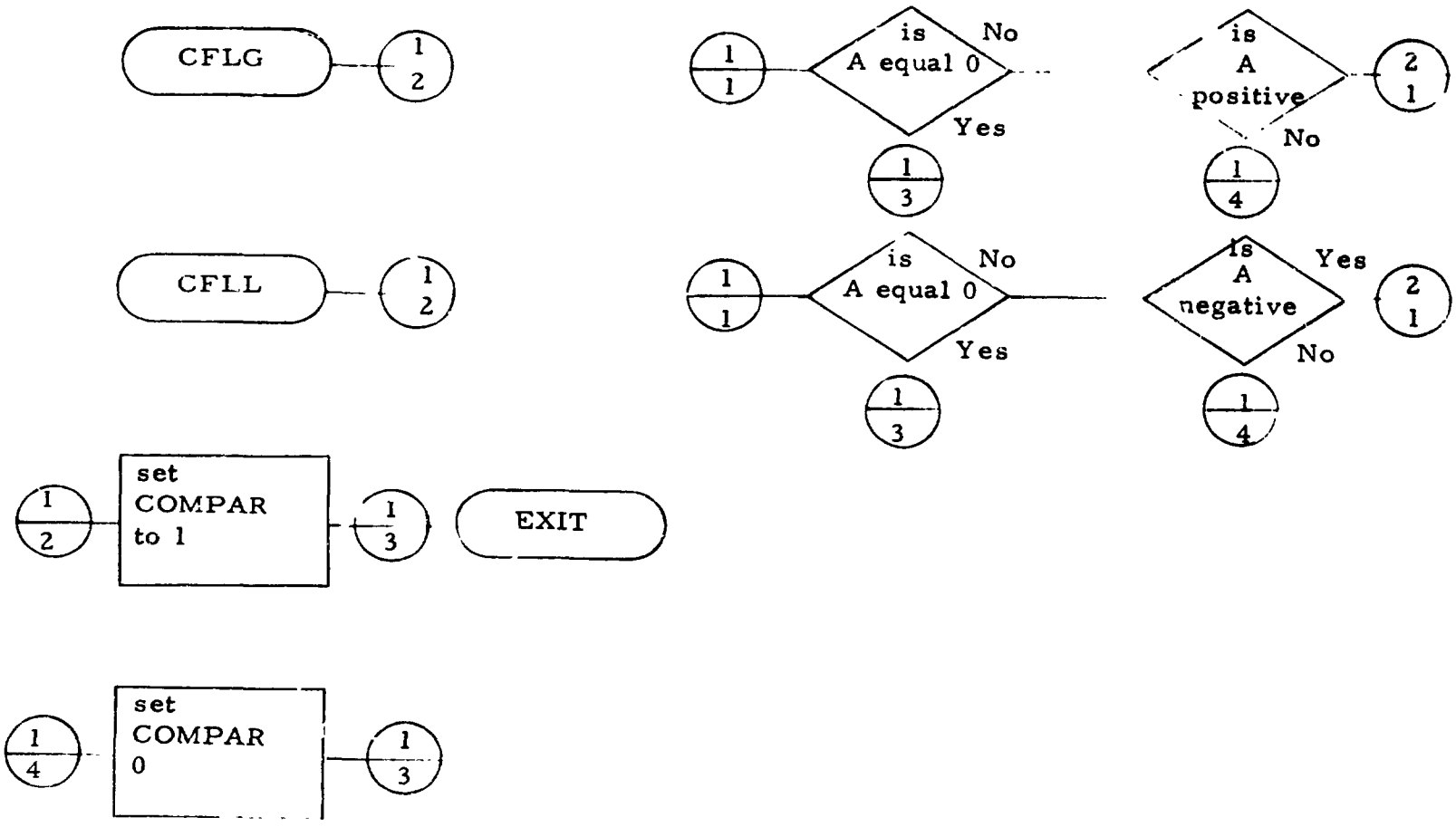
2. Abstract: A comparison of FLACC and an operand is made for both programs. Each program has its own conditional test (greater than or less than). If the condition is satisfied the compare indicator COMPAR is set true. If the condition is not satisfied COMPAR is set false. If the two numbers are equal COMPAR is not changed.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-3
6. Program Descriptions: Each program has its own entry point. Both do a return jump to a subroutine to test the two numbers. This subroutine loads FLACC in the A register and the operand in the Q register. A check is made of the signs of A and Q. If A is positive and Q negative, A is returned with a 1. If A is negative and Q positive, A is returned with a -1. If A and Q are alike in sign, Q is subtracted from A and the result tested. If the result is zero return to the calling program. If it is positive set A to 1 and if negative set A to -1. Return to calling program. The calling program tests the A register for its own particular condition and sets COMPAR accordingly true, false, or no change.

7. **Inputs:** /BLOCK/INDEX/OPN/

8. **Outputs:** COMPAR is set true, false or is not changed.

9. **Call Sequence:**

```
RTJ     CFLC or
RTJ     CFLL or
```



Floating Point Compare

Figure 10-3

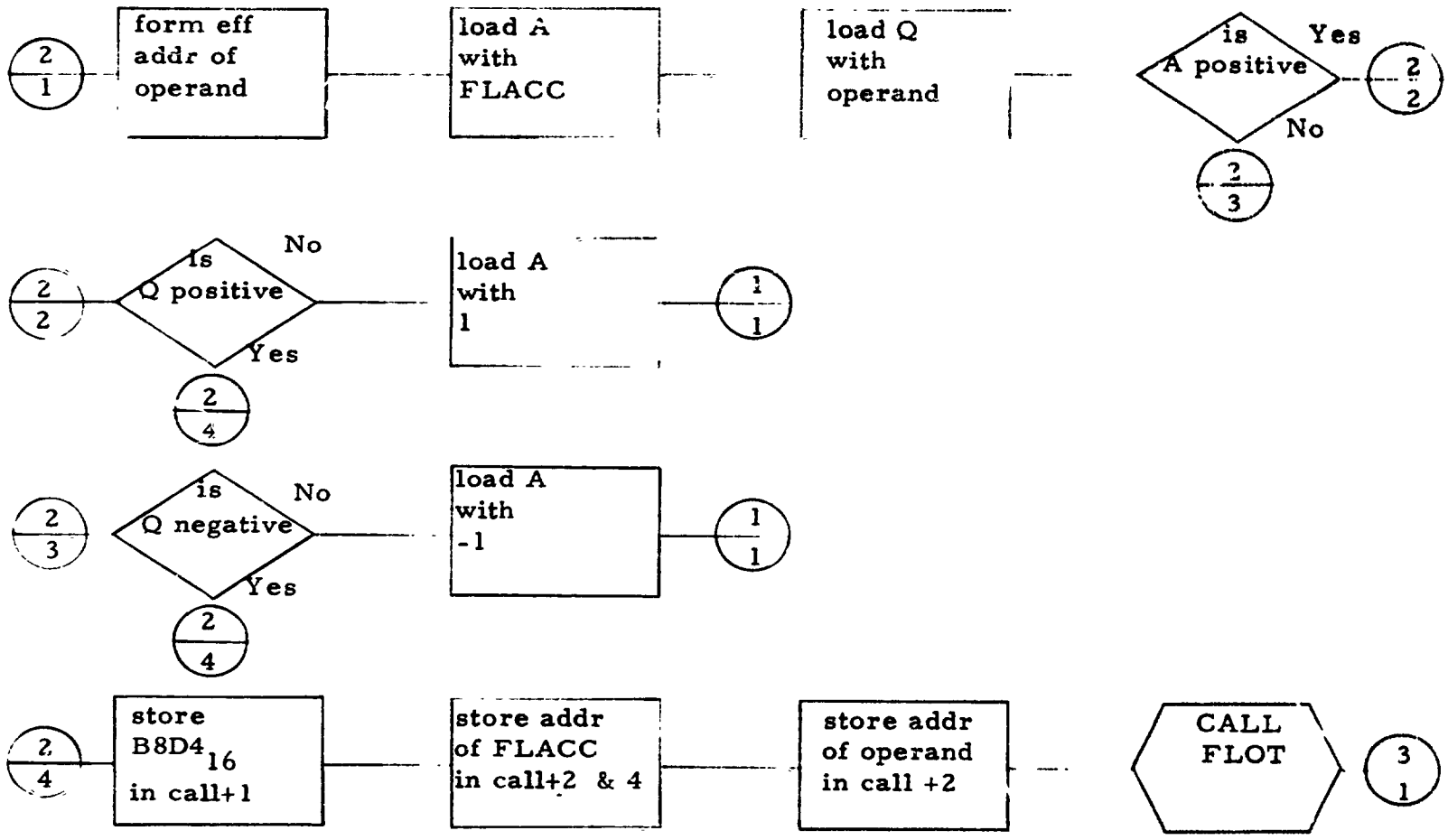


Figure 10-3 (Cont'd)

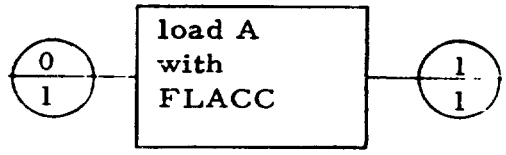


Figure 10-3 (Cont'd)

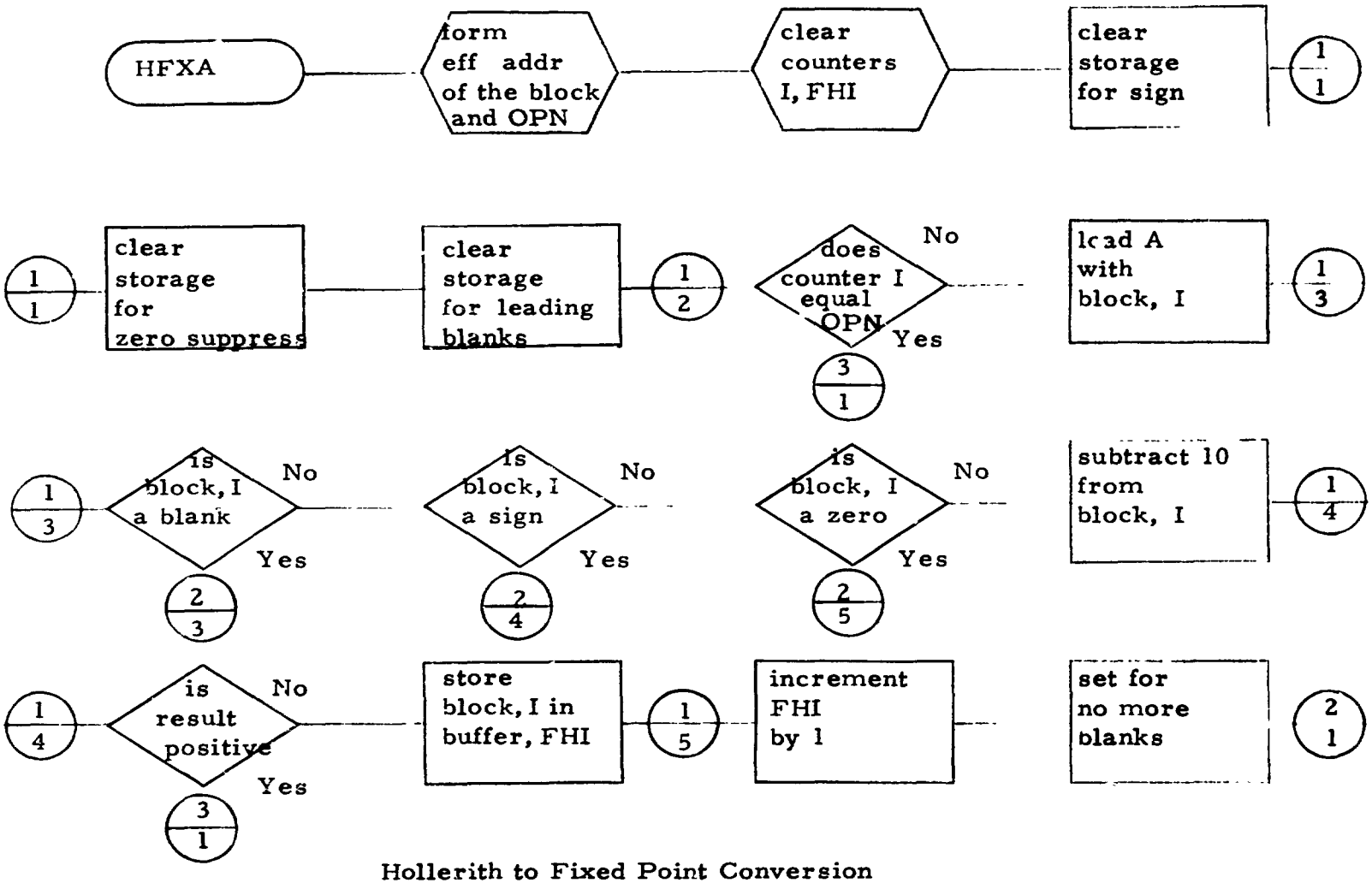
10.5 HOLLERITH TO FIXED POINT CONVERSION

1. Program Name: HFXA - Hollerith to fixed point conversion.
2. Abstract: Program HFXA converts a string of words containing decimal numbers to one fixed point numeral.
3. Operational Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-4
6. Program Description: On entry of the program the effective address of the block of data to be converted is formed and the number of words (OPN) in the block is saved. Each word is first checked to see if it is a blank. If it is a blank it is checked to see if it is a leading blank. If it is a leading blank it is discarded. If it is not, the blank is taken to mean end of data. A test for sign is made next. Each time a sign is encountered, it is saved and the last saved sign is used as the sign of the fixed point. Next we check on the validity of the word. It must be a number between 0 and 9, otherwise, all other data inputs is taken to mean end of data. If the number is a leading zero it is discarded, otherwise, it is a valid number. As the fixed point number is built, the number of digits processed is tested so that if more than 5 of them will be formed overflow occurs. Another check for overflow is made when the fixed point number is exactly 5 digits long. The number must be in the range -32767 and +32767. If overflow occurs FXOVFL is set to 1. The final result is stored in FXACC. Exit.

7. Inputs: /BLOCK/INDEX/OPN/

8. Outputs: A fixed point number is stored in FXACC. On overflow FXOVFL is set to one.

9. Call Sequence: RTJ HFXA



Hollerith to Fixed Point Conversion

Figure 10-4

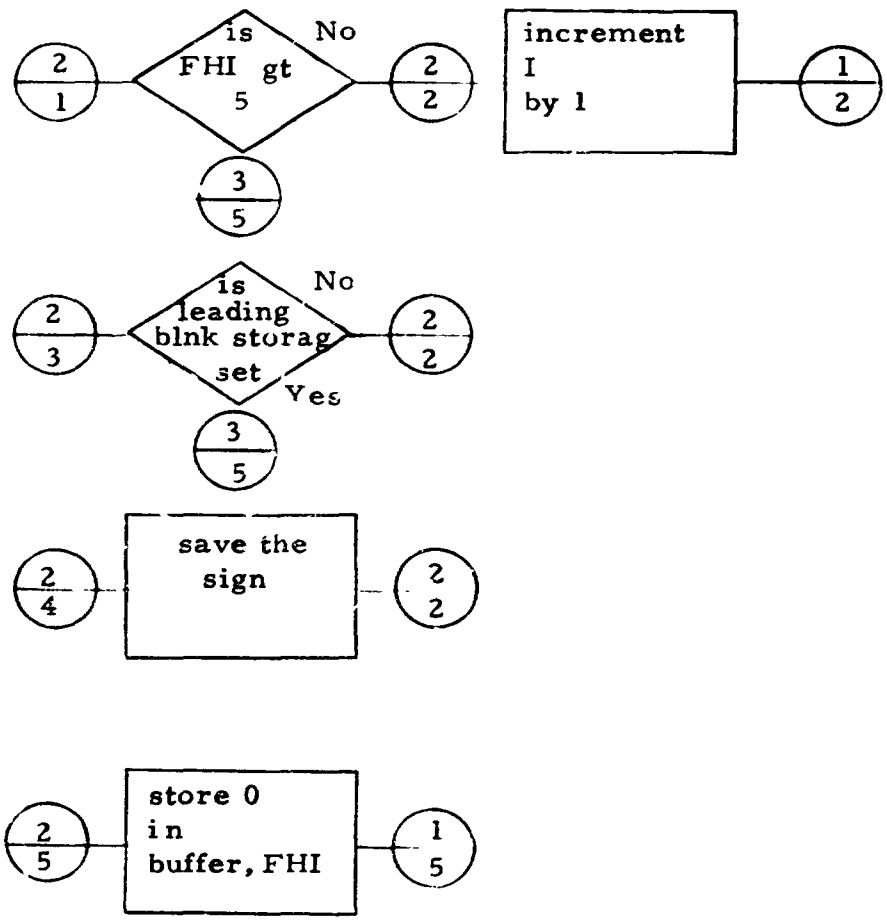


Figure 10-4 (Cont'd)

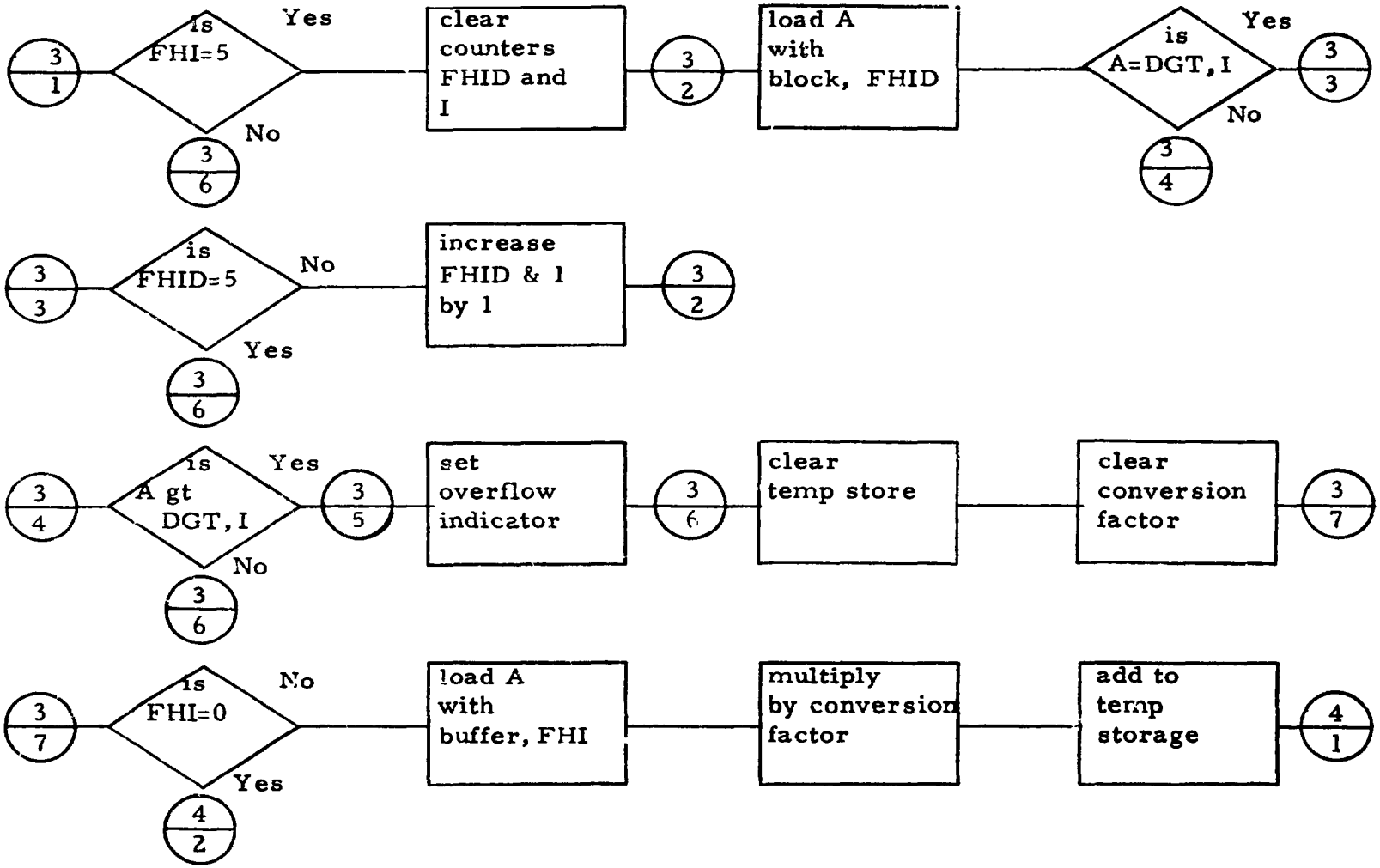


Figure 10-4 (Cont'd)

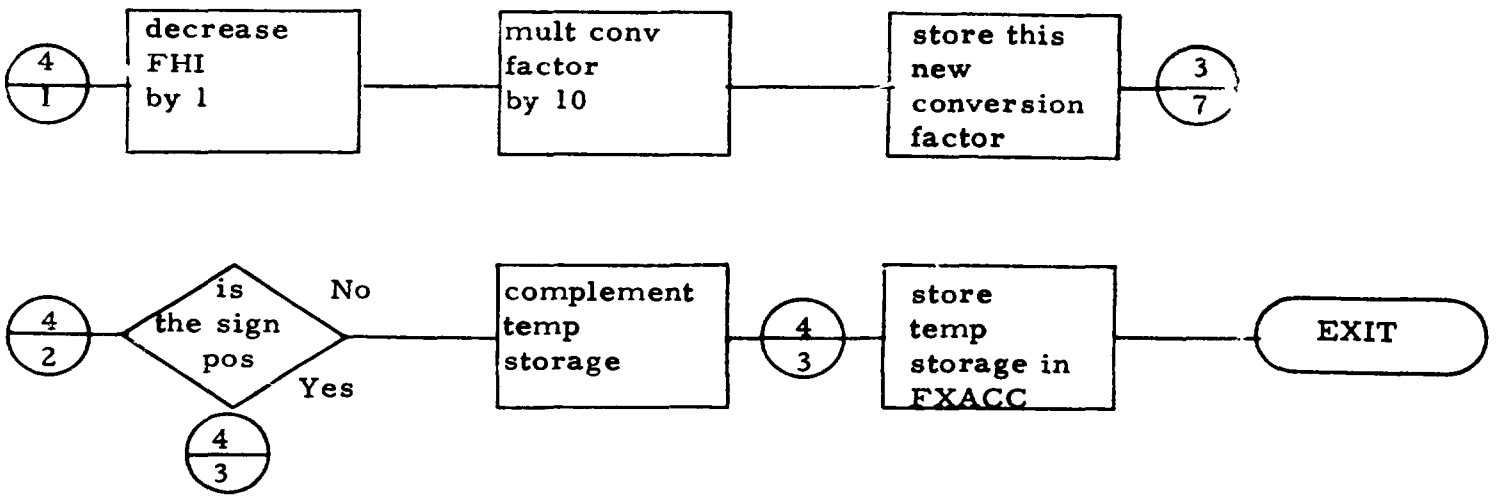
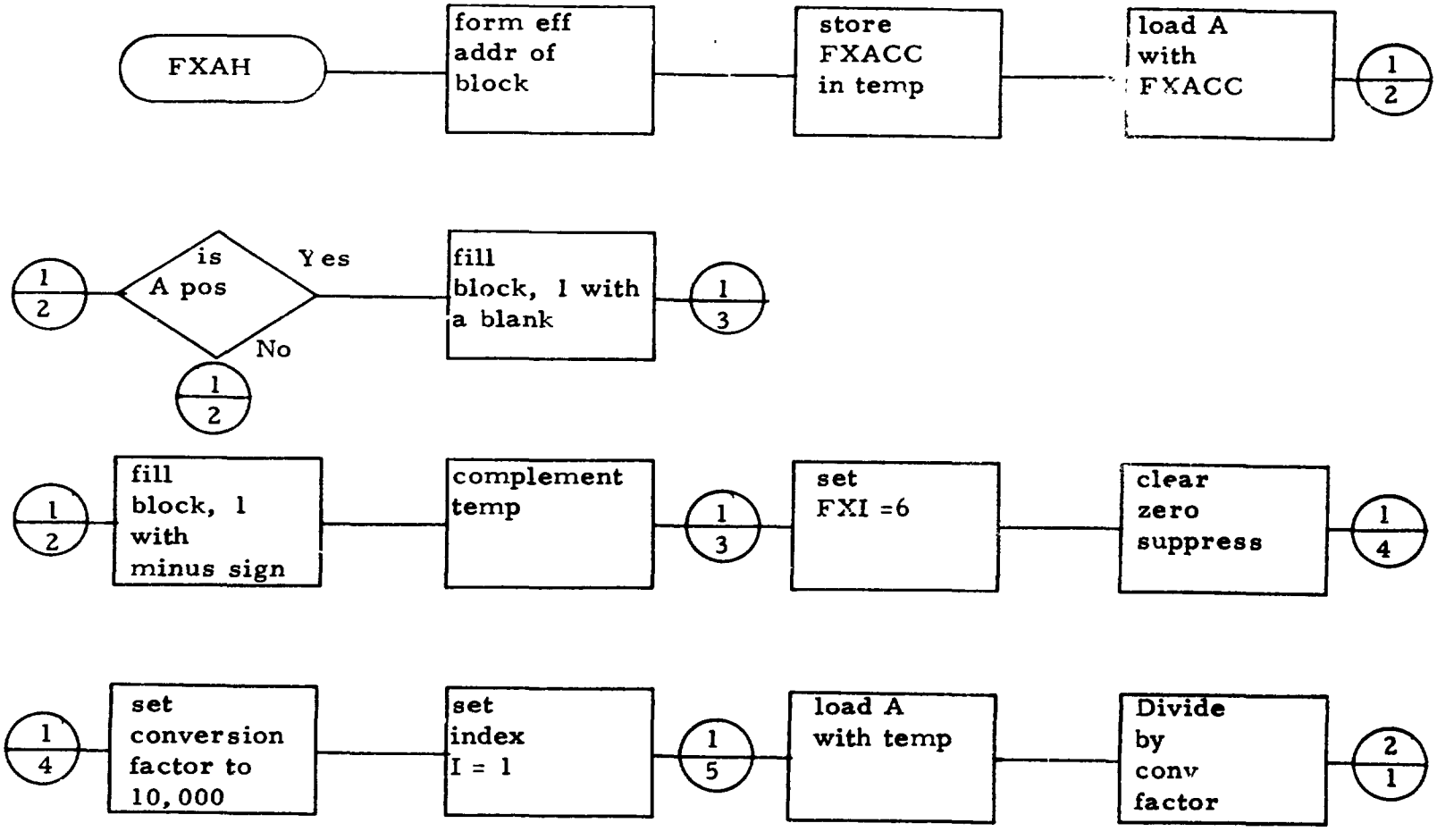


Figure 10-4 (Cont'd)

10.6 FIXED POINT TO HOLLERITH DECIMAL CONVERSION

1. Program Name: FXAH - Fixed Point to hollerith conversion.
2. Abstract: FXACC is converted to hollerith decimal digits and stored in a given block, OPN words long. The first word will contain the sign (a blank if positive) and the remaining words (up to 5 more) the digits. If OPN is specified longer than is needed, the remaining words of the block are blank filled.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-5.
6. Program Description: On entry of the program the effective address of the block to be filled with data is saved. The OPN count is also retained. The sign of FXACC is converted to hollerith format and stored in the first word of the block. (If it was positive a blank is used). As the fixed point number is converted leading zeros which appear are suppressed. All other digits (including significant zeroes) are stored one at a time in succeeding words of the given block. As a final step, OPN is tested against the number of words used in the block. If there are more words to be processed in the block, they are blank filled.
7. Inputs: /BLOCK/INDEX/OPN/
8. Outputs: A given block is filled with hollerith characters converted from a fixed point number.
9. Call Sequence: RTJ FXAH



Fixed Point to Hollerith Conversion

Figure 10-5

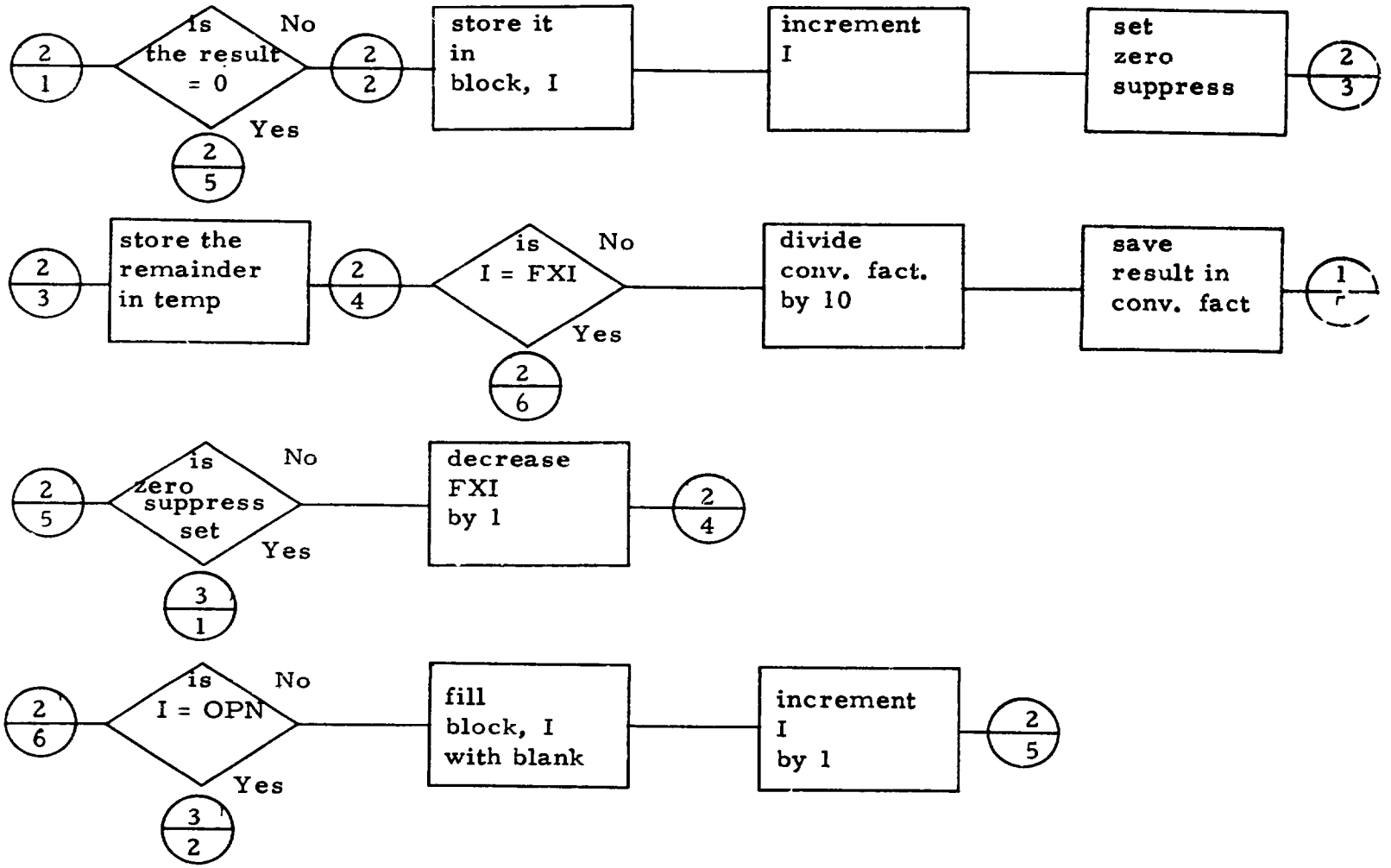


Figure 10-5 (Cont'd)

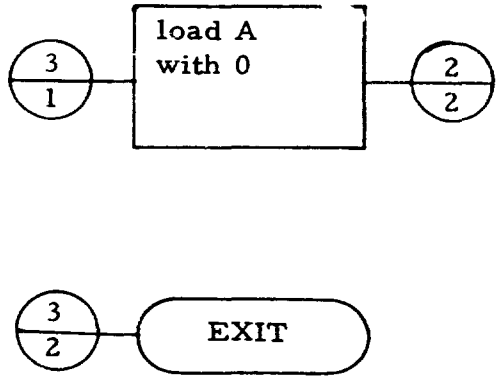
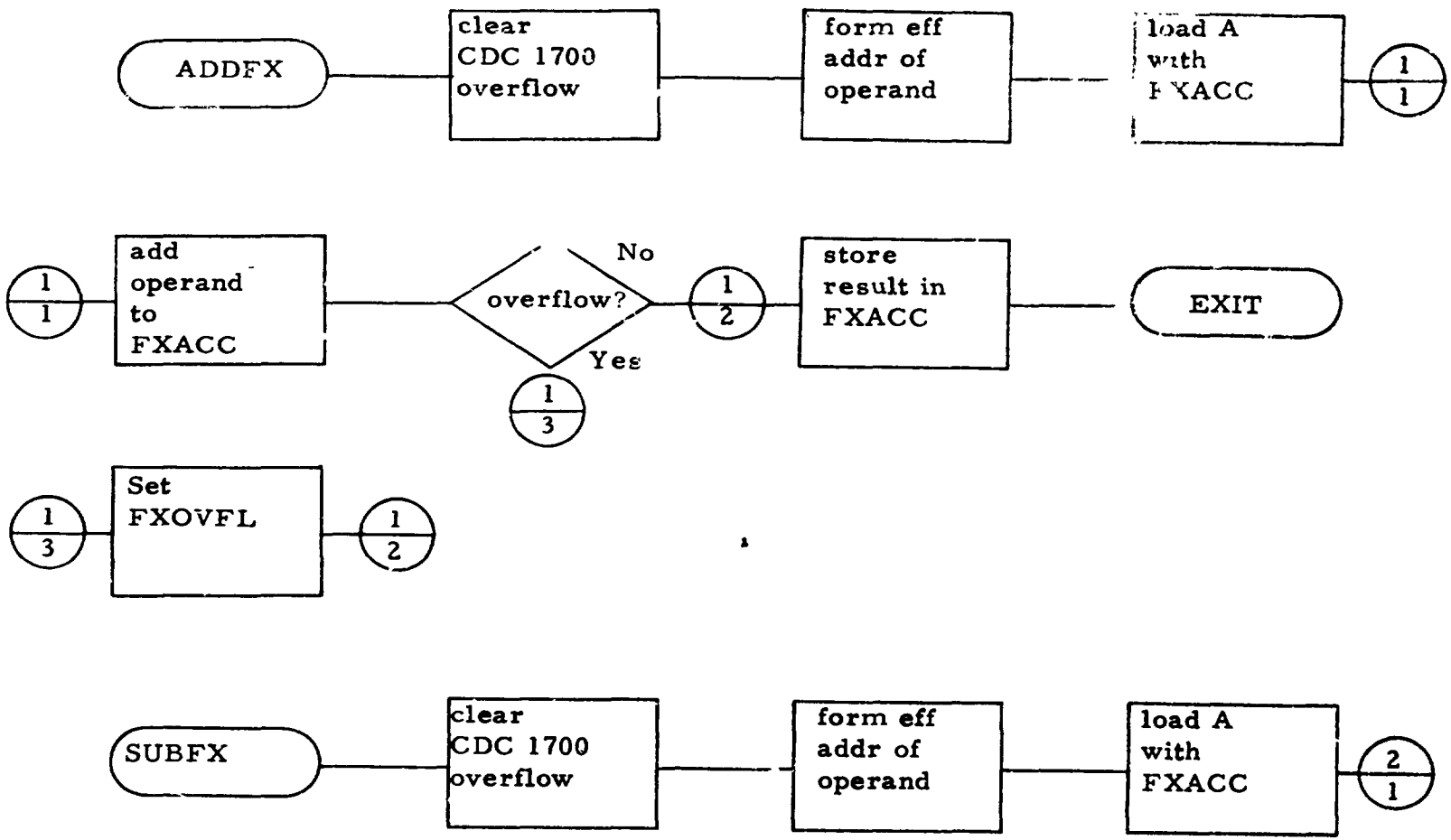


Figure 10-5 (Cont'd)

10.7 FIXED POINT ARITHMETIC

1. Program Names: ADDFX - Fixed point add
   SUBFX - Fixed point subtract
   MUL - Fixed point multiply
   DIV - Fixed point divide
2. Abstract: The named programs perform their respective fixed point arithmetic between FXACC an ' the given operand. In case of overflow the overflow indicator is set.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-6.
6. Program Description: Each program has its own entry point. The effective address of the operand is formed and stored. The CDC 1700 overflow is cleared before continuing. Each program performs its respective arithmetic function between FXACC and the operand. (DIV first checks for division by zero. If so, the FXACC is filled with $7FFF_{16}$, FXOVFL is set and exit occurs.) The CDC 1700 overflow indicator is tested. If overflow has occurred, the fixed point overflow indicator FXOVFL is set. Exit.
7. Inputs: /BLOCK/INDEX/OPN/
8. Outputs: Result of the operation is stored in FXACC. FXOVFL is set on overflow.
9. Call Sequence:
```
RTJ     ADDFX or
RTJ     SUBFX or
RTJ     MUL or
RTJ     DIV
```



Fixed Point Arithmetic

Figure 10-6

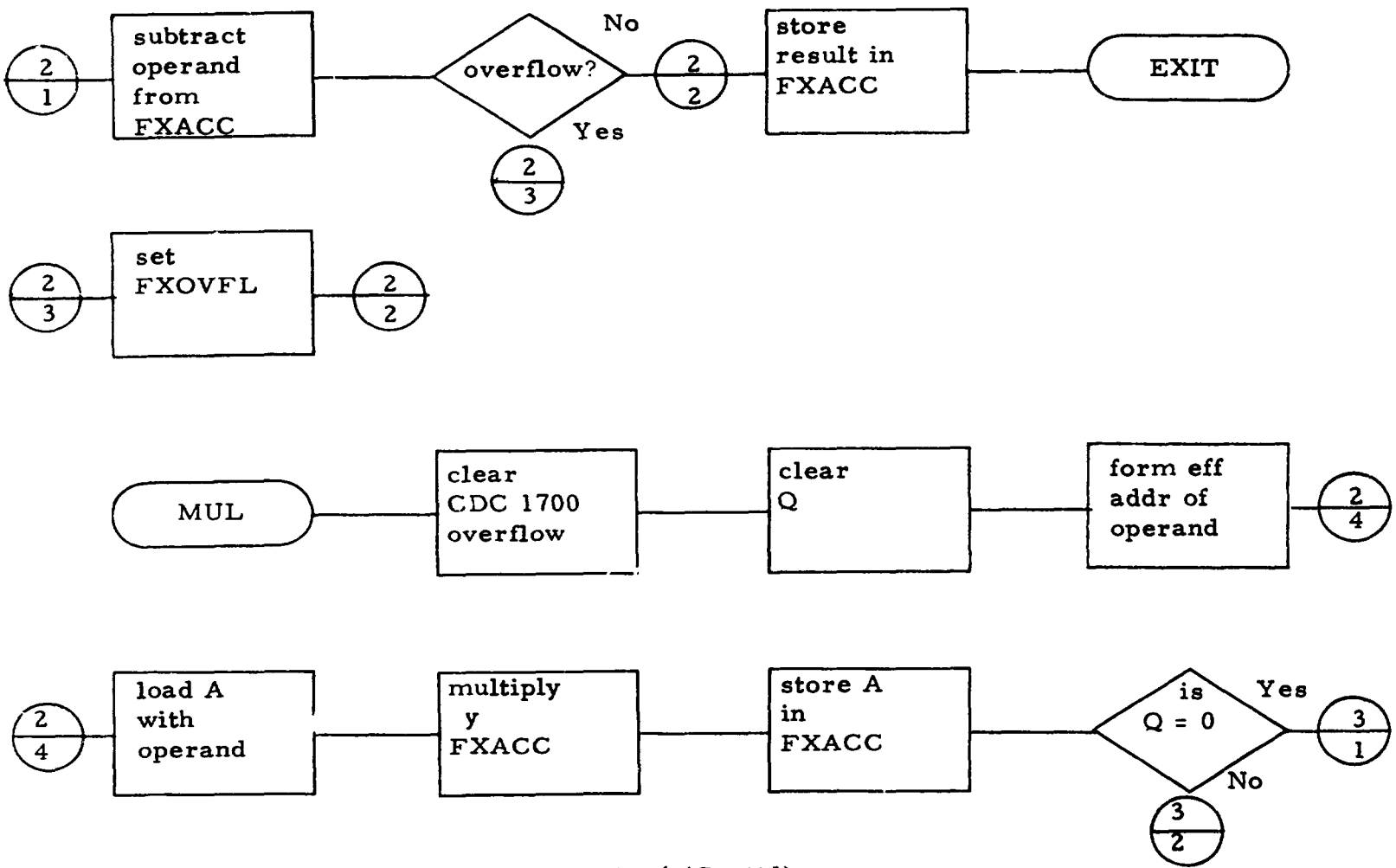


Figure 10-6 (Cont'd)

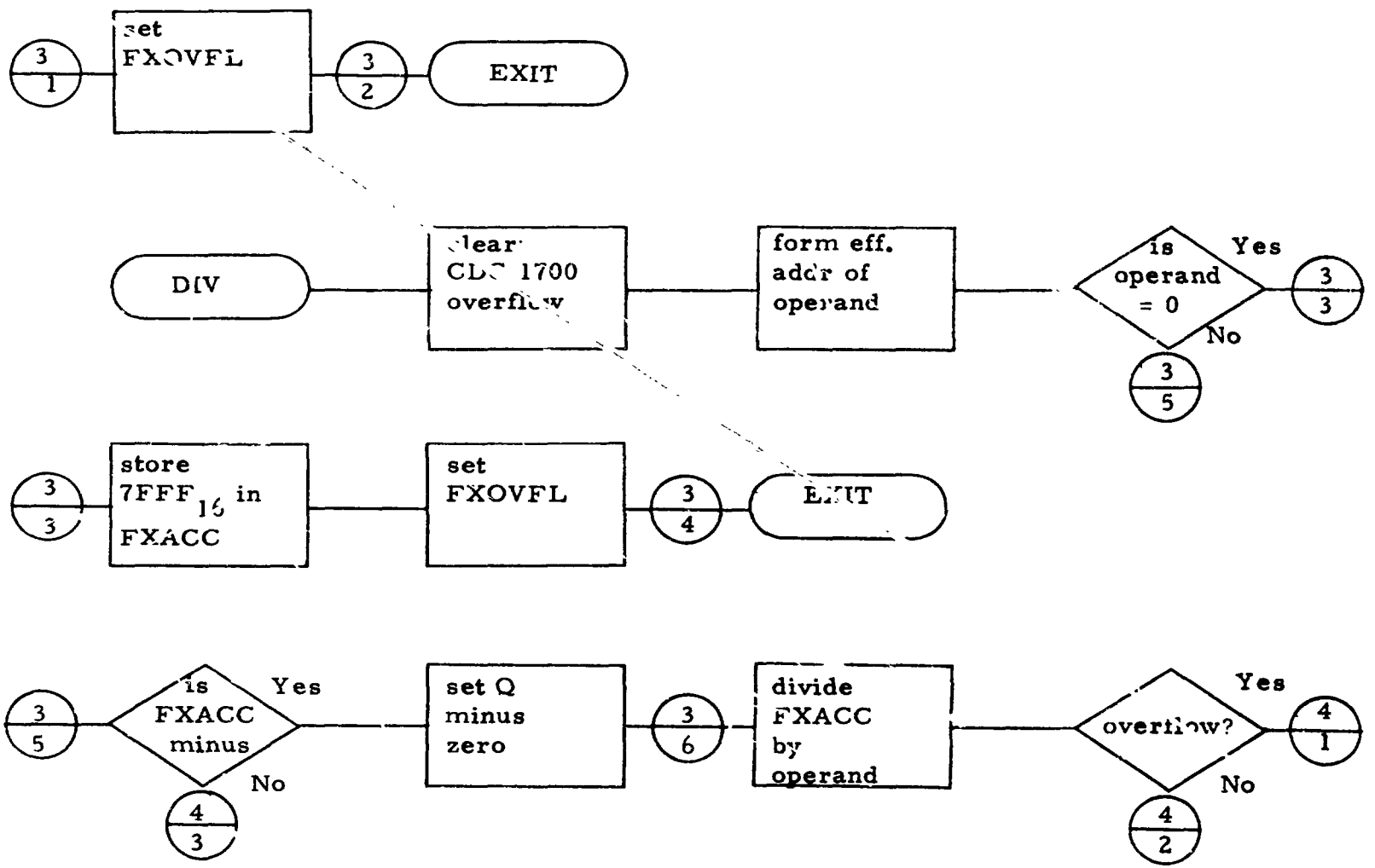


Figure 10-6 (Cont'd)

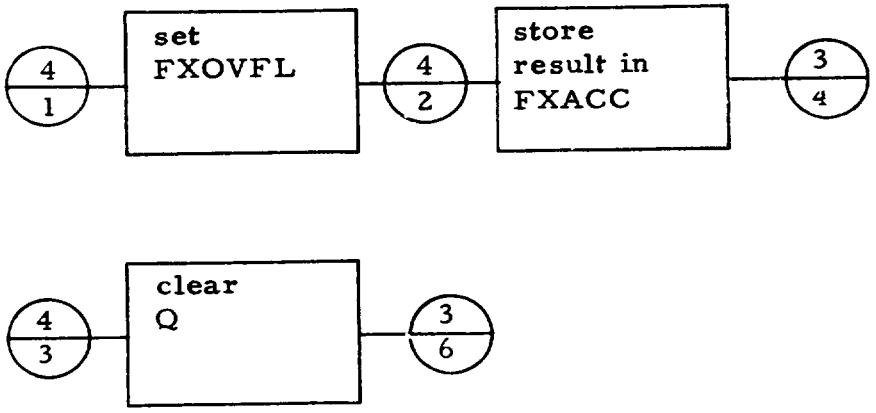
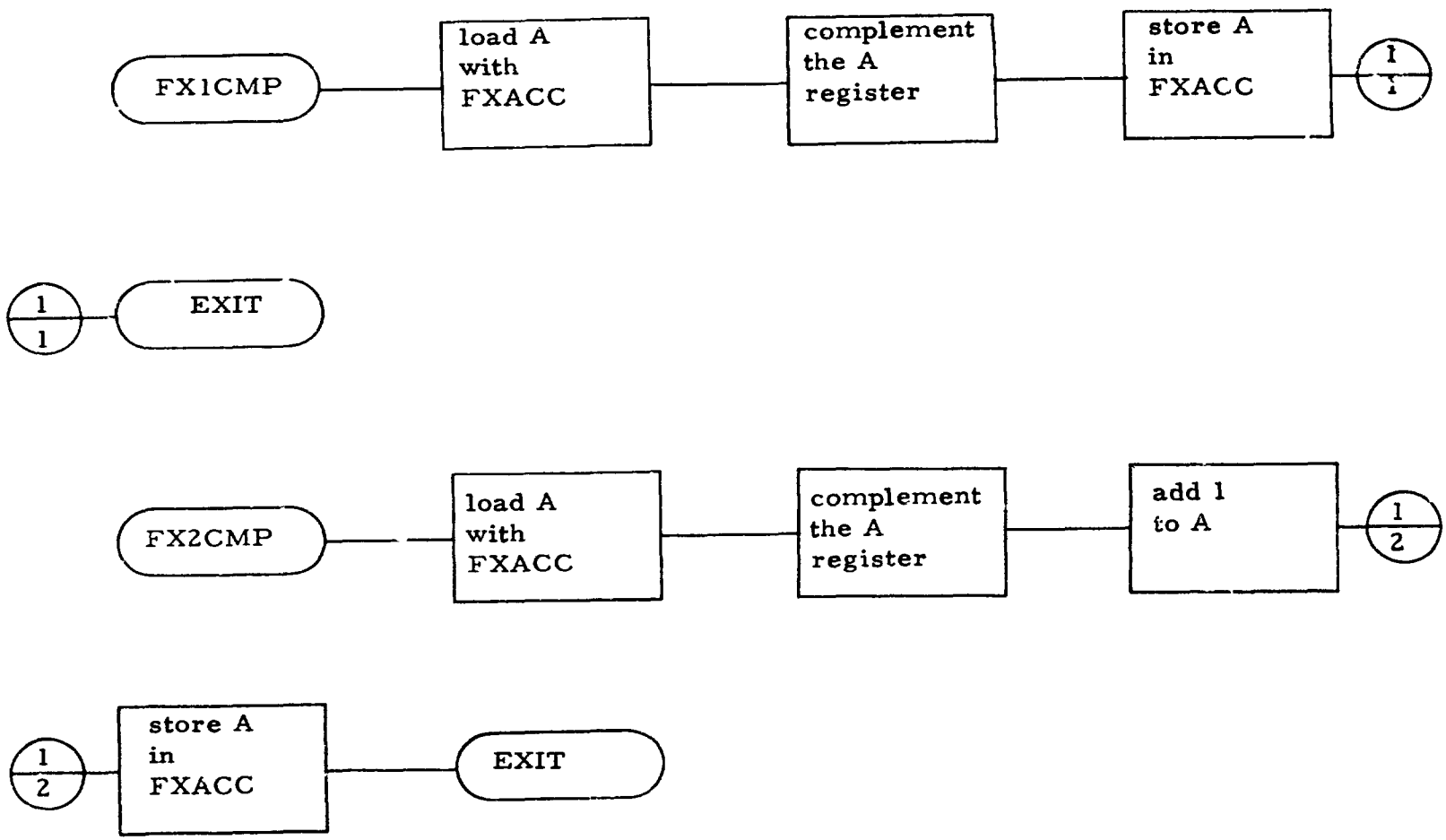


Figure 10-6 (Cont'd)

10.8 FIXED POINT COMPLEMENTATION

1. Program Names: FX1CMP - ones complement
   FX2CMP - twos complement
2. Abstract: Convert the contents of FXACC to its complement form. This may be either ones or twos complement.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-7
6. Program Description: Each program has its own entry point. Both load the A register with the contents of FXACC. The A register is complemented. (FX2CMP also adds 1 to the A register after the complement operation.) The result is stored in FXACC. Exit
7. Inputs: None
8. Outputs: Result of the operation is stored in FXACC.
9. Call Sequence:

```
RTJ     FX1CMP or
RTJ     FX2CMP
```



Fixed Point Complement

Figure 10-7

10.9 FIXED POINT COMPARE

1. Program Names: CMPGFX - Fixed point compare if greater than operand.

   CMPEFX - Fixed point compare if equal to operand

   CMPLFX - Fixed point compare if less than operand

2. Abstract: A comparison of FXACC and an operand is made by all three programs. Each program has its own conditional test (greater than, equal to or less than). If the condition is satisfied, the compare indicator COMPAR is set true. If the condition is not satisfied, COMPAR is set false.

3. Operating Requirements: CDC 1700.

4. Language: CDC 1700 Assembler

5. Functional Flow Charts: See Figure 10-8.

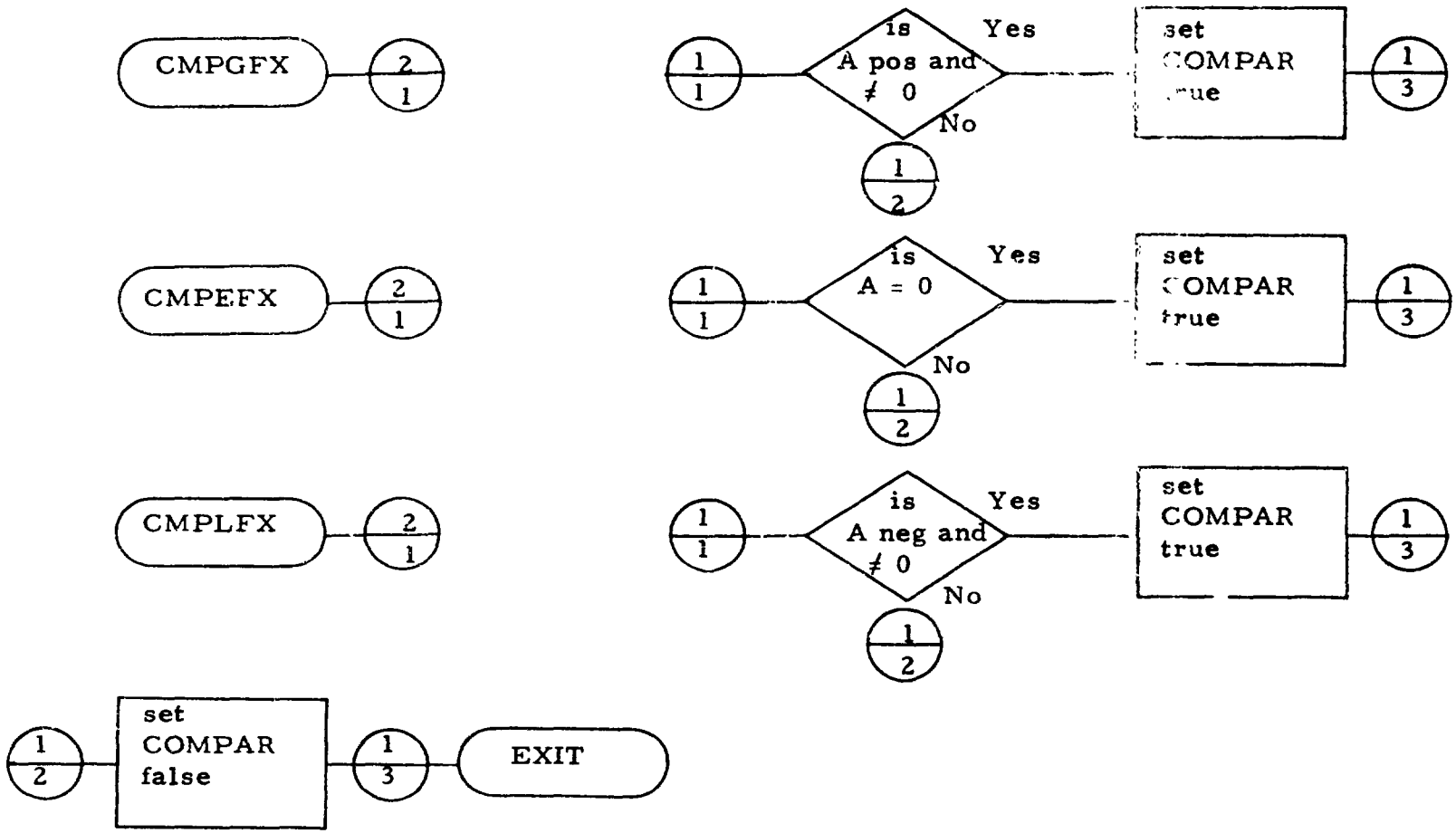
6. Program Description: Each program has its own entry point. Each do a return jump to a subroutine to test the two numbers. This subroutine loads FXACC in the A register and the operand in the Q register. A check is made of the signs of A and Q. If A is positive and Q is negative, A is returned with a 1. If A is negative and Q positive, A is returned with a -1. If A and Q are alike in sign Q is subtracted from A and the result is return to the calling program in the A register. The calling program tests the A register for its own particular condition and sets COMPAR accordingly true or false.

7. Inputs: /BLOCK/INDEX/OPN/

8. Outputs: COMPAR is set true or false

9. Call Sequence:

```
RTJ     CMPGFX or
RTJ     CMPEFX or
RTJ     CMPLFX
```



Fixed Point Compare

Figure 10-8

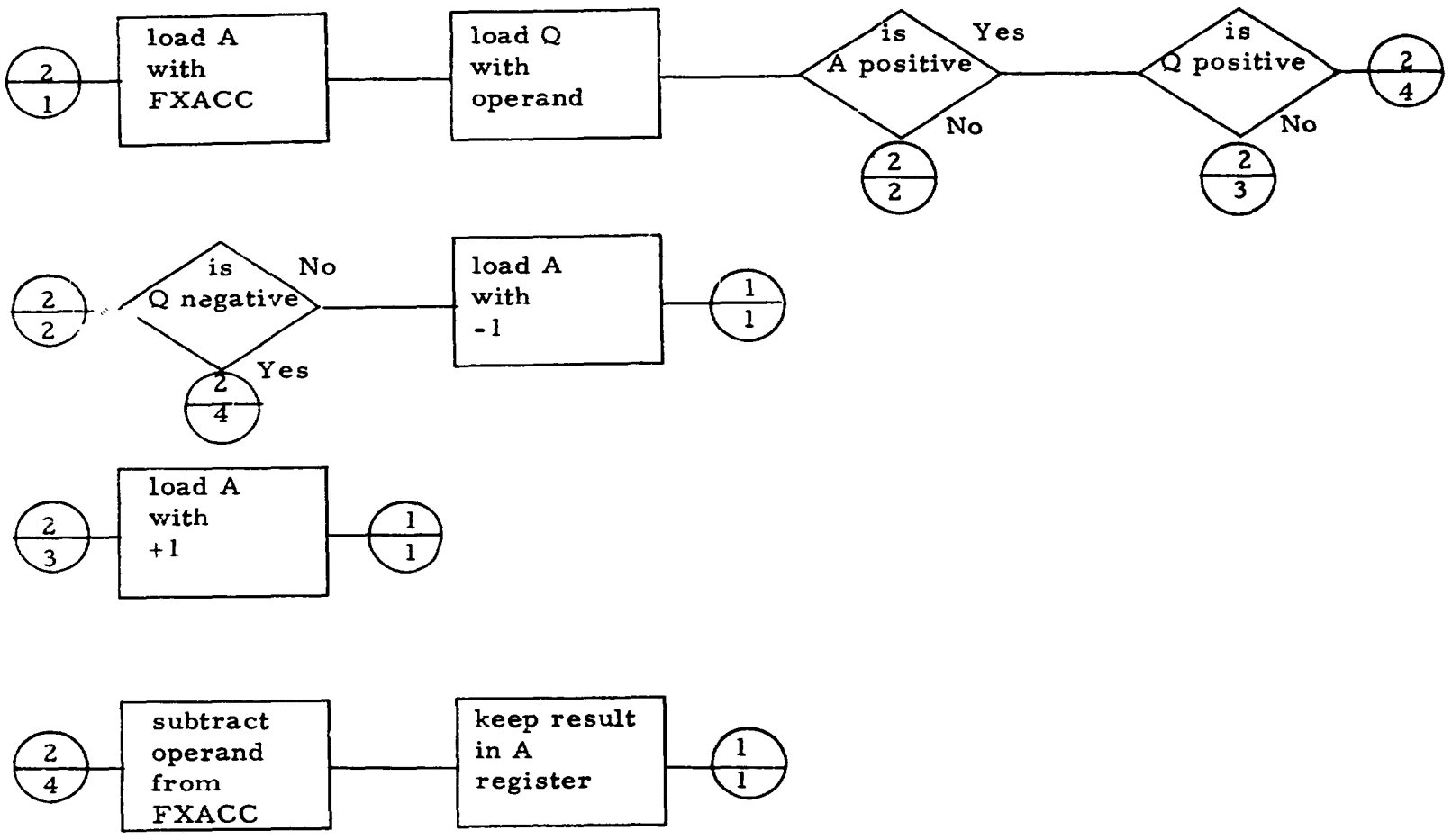
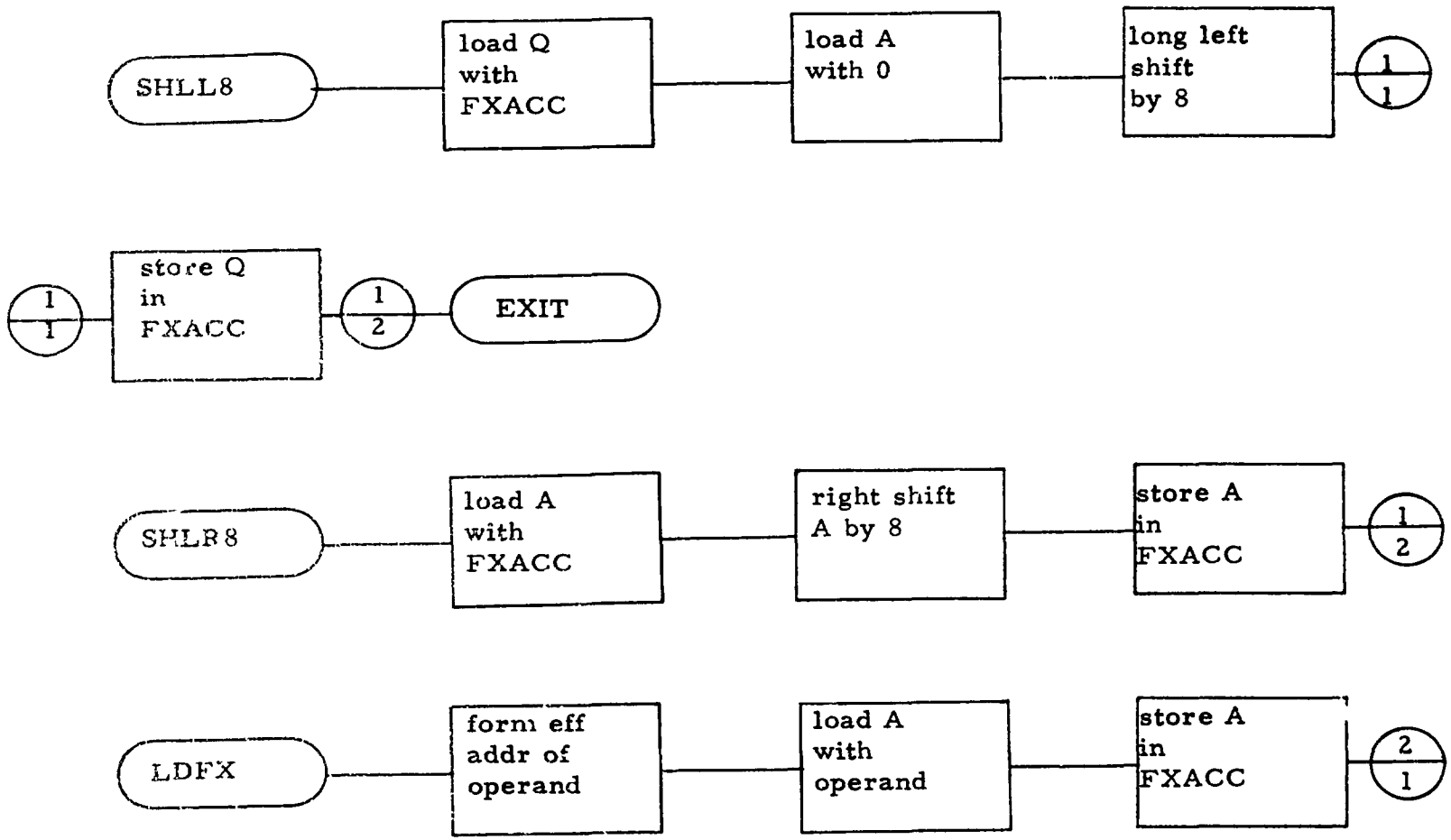


Figure 10-8 (Cont'd)

10.10 FIXED ACCUMULATOR OPERATIONS

1. Program Name: SHLL8 - Shift Logical Left 8 the Fixed Accumulator
   SHLR8 - Shift Logical Right 8 the Fixed Accumulator
   LDFX - Load the Fixed Accumulator
   STFX - Store the Fixed Accumulator

2. Abstract: The shift programs simply shift the accumulator left or right by 8 with no end around. The right shift is sign extended. The load and store simply load an operand into FXACC and store FXACC into an operand.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembly Language
5. Functional Flow Charts: See Figure 10-9
6. Program Description: SHLL8 loads Q with FXACC and A with 0. A long left shift by 8 is done and Q is stored in FXACC. This allows for no end around. SHLR8 loads A with FXACC, right shifts with sign extended by 8 and stores the result in FXACC. Both LDFX and STFX form the effective address of the operand. LDFX loads FXACC with the operand. STFX stores FXACC in the operand.
7. Inputs: /BLOCK/INDEX/OPN/
8. Outputs: Specified function result i stored in FXACC.
9. Call Sequence:
```
RTJ   SHLL8 or
RTJ   SHLR8 or
RTJ   LDFX or
RTJ   STFX
```



Fixed Accumulator Operations

Figure 10-9

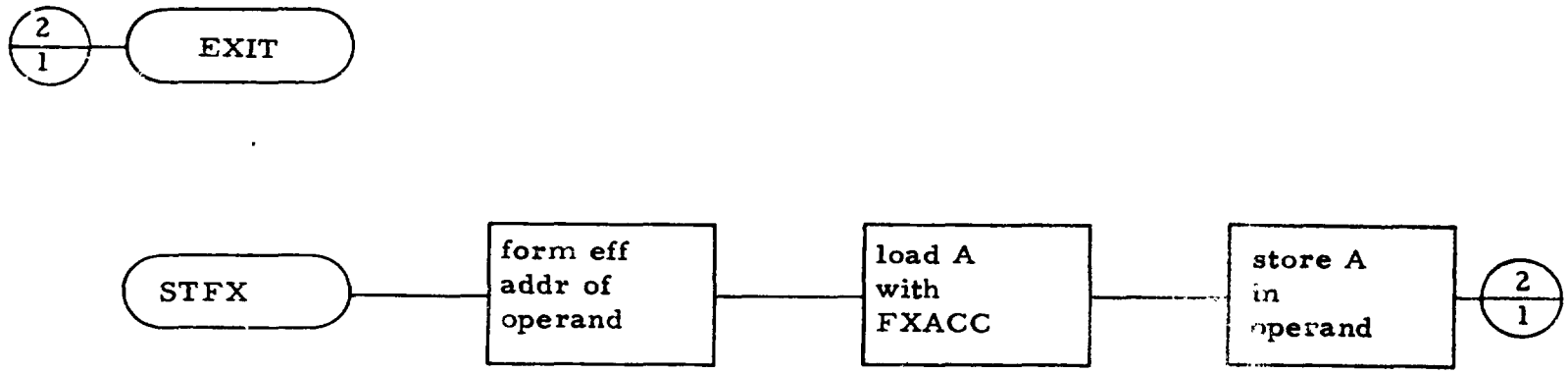


Figure 10-9 (Cont'd)

10.11 FIXED POINT BLOCK SEARCH

1. Program Names: SRCHNH - Search numeric high
   SRCHEQ - Search equal
   SRCHNL - Search numeric low
   SRCHNQ - Search not equal

2. Abstract: All four programs search a block of data for a fixed point number which satisfies the specified condition. That is, the program tests a block of data against FXACC. The first time that FXACC is numerically higher than any number in the block of data the condition is satisfied for SRCHNH. The first time that FXACC is equal to any number in the block of data the condition is satisfied for SRCHNL. The first time that FXACC is not equal to number in the block of data the condition is satisfied for SRCHNQ.

3. Operating Requirements: CDC 1700

4. Language: CDC 1700 Assembler

5. Functional Flow Charts: See Figure 10-10

6. Program Description: Each program has its own entry point. A test condition is built into a location of the search subroutine common to the four programs. SRCHNH tests the A register with an SAN instruction, SRCHEQ with an SAZ, SRCHNL with an SAP and SRCHNQ with an SAN. A return jump is made to the subroutine. Index Q is cleared and the effective address of the block of data is formed. The block is indexed by Q and a search is performed until the test condition is met. If the test is satisfied return to the calling program with the contents of index Q in FXACC. If the test is not met

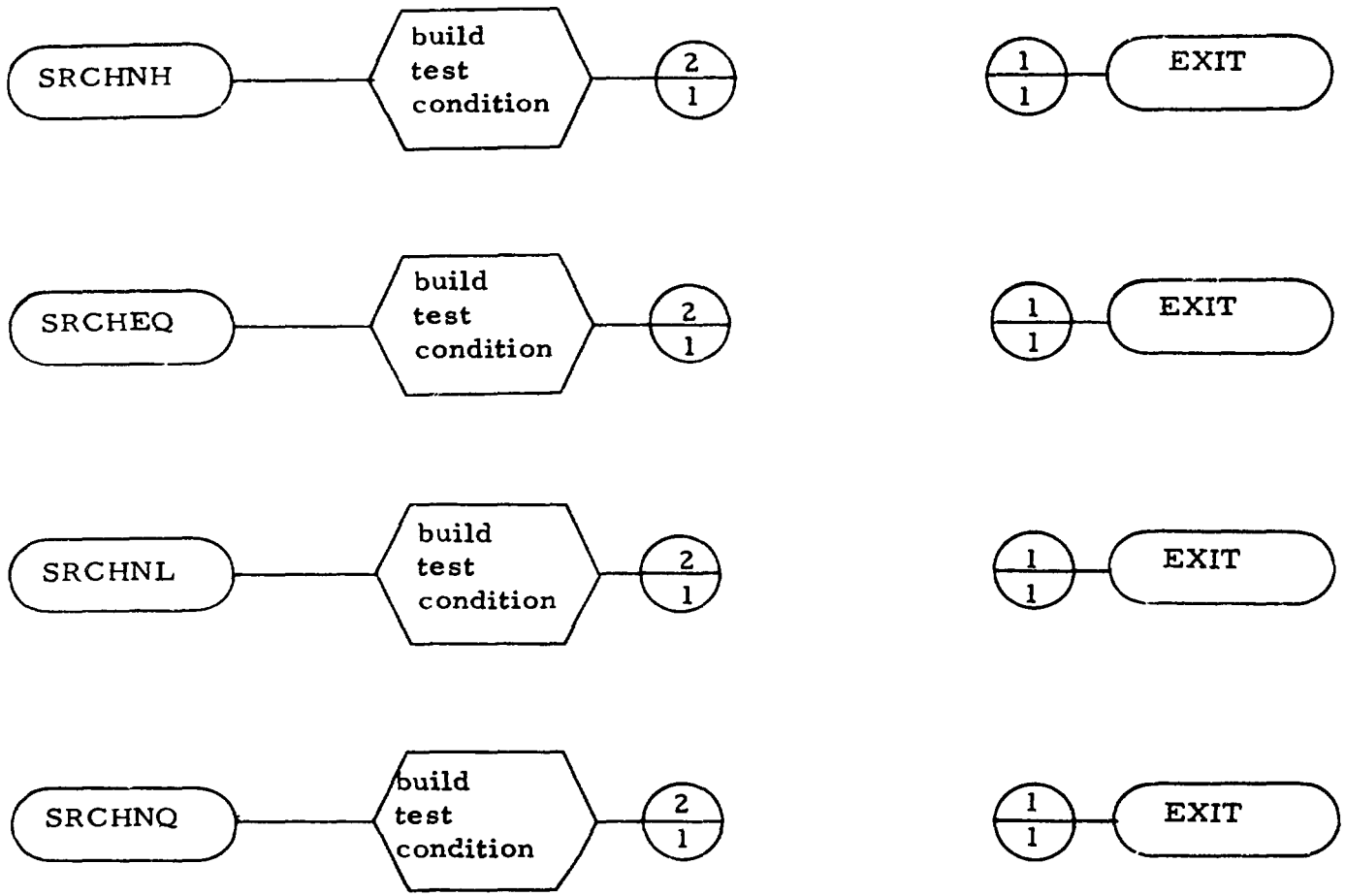
and the whole block has been searched, a return to the calling program is performed with FXACC containing -1.

7. Inputs: /BLOCK/INDEX/OPN/

8. Outputs: FXACC will contain the value of the block index when the test condition was met. FXACC will contain -1 if the test condition was not met throughout the block.

9. Call Sequence:

```
RTJ     SRCHNH or
RTJ     SRCHEQ or
RTJ     SRCHNL or
RTJ     SRCHNQ
```



Fixed Point Block Search

Figure 10-10

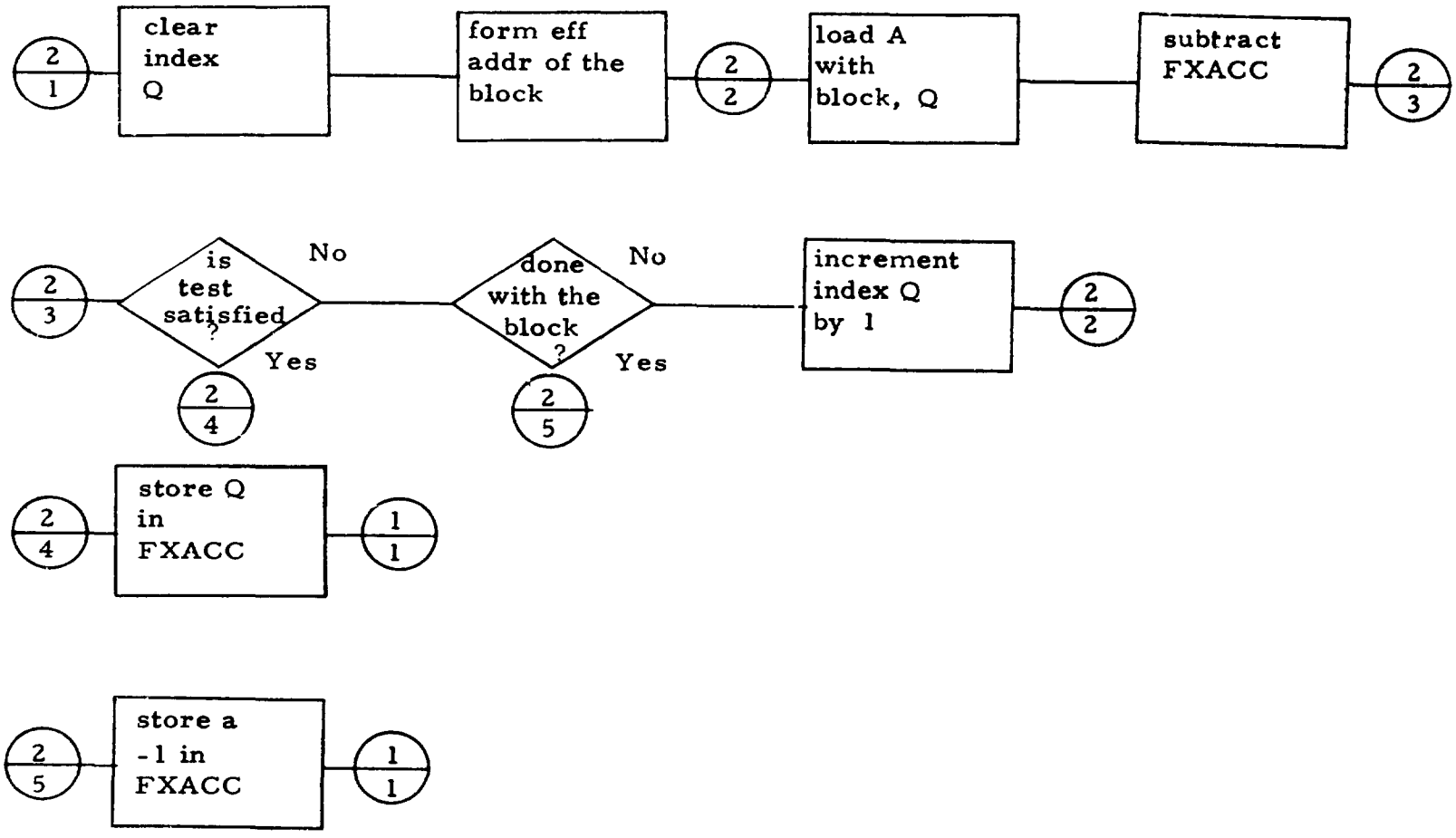
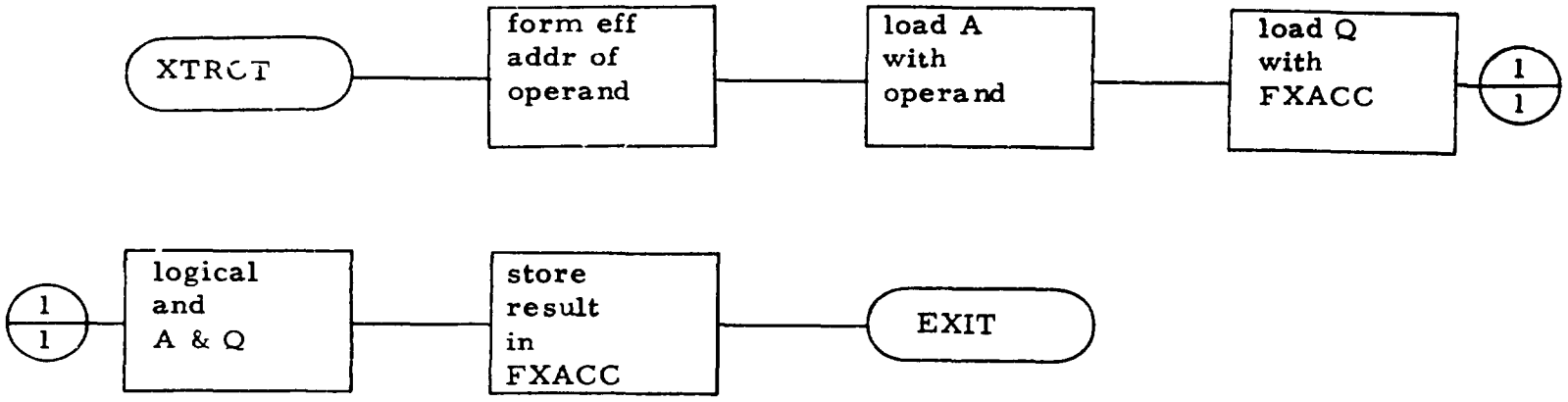


Figure 10-10 (Cont'd)

10.12 FIXED POINT EXTRACT

1. Program Name: XTRCT - Extract (logical and) fixed point
2. Abstract: XTRCT performs a "logical and" between FXACC and an operand.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-11
6. Program Description: On entry to the program the effective address of the operand is formed and stored in the A register. The Q register is loaded with FXACC. A "logical and" between A and Q is performed and the result stored in FXACC. Exit.
7. Input: /BLOCK/INDEX/OPN/
8. Outputs: Result of the operation in the FXACC.
9. Call Sequence: RTJ XTRCT



Fixed Point Extract

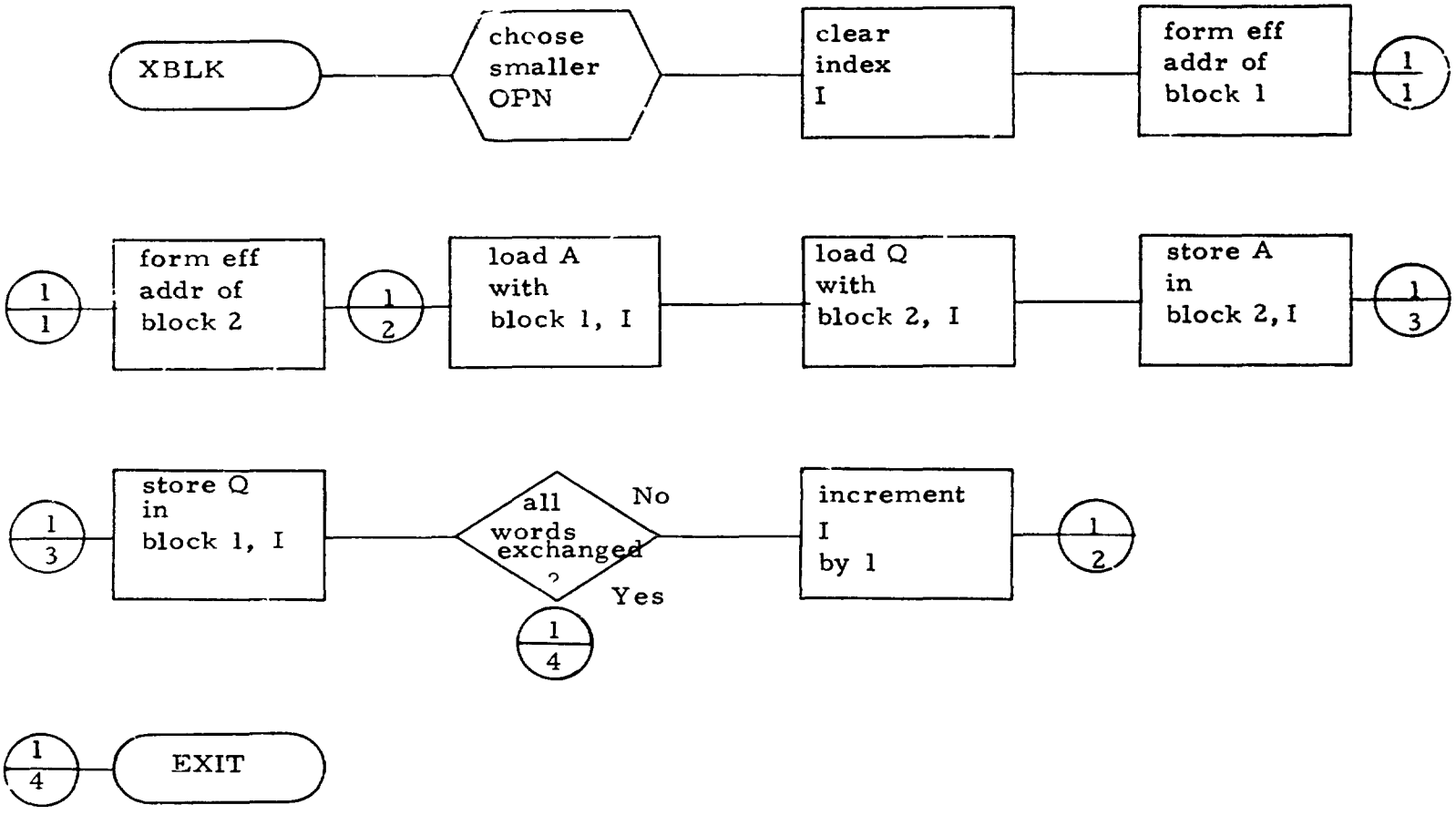
Figure 10-11

10.13 FIXED POINT BLOCK TRANSFER

1. Program Name: XBLK - Block Transfer Exchange
2. Abstract: The function of this program is to exchange the contents of two strings (blocks) of data. The operation is performed in ascending order beginning with word zero of the specified block. The word count is as long as the shortest word count of the two blocks of data.
   NOTE: The exchange is performed one word at a time. A floating point number could be considered a block two words long and exchanged with another block. Since no conversion is attempted, this could result in unexpected trouble for the user. It is recommended that this be not attempted.
3. Operational Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-12
6. Program Description: On entry of the program, OPN, the number of words in the block, is calculated. If the two blocks are of different length, OPN of the shorter block is chosen as OPN for both. The index I into the block is set to zero. Next, the effective addresses of the blocks are formed and stored. The A register is loaded with the contents of the first block address plus the index I. Q is loaded with the contents of the second block address plus the index I. The A register

is stored in the second block address plus the index I and the Q register is stored in the first block address plus the index I. If OPN number of words have not been exchanged, index I is increased by 1 and the exchange process is repeated. If OPN number of words have been exchanged, exit.

7. Inputs: /BLOCK/INDEX/OPN/BLOCK/INDEX/OPN/
8. Outputs: Two blocks are exchanged word for word.
9. Call Sequence: RTJ XBLK

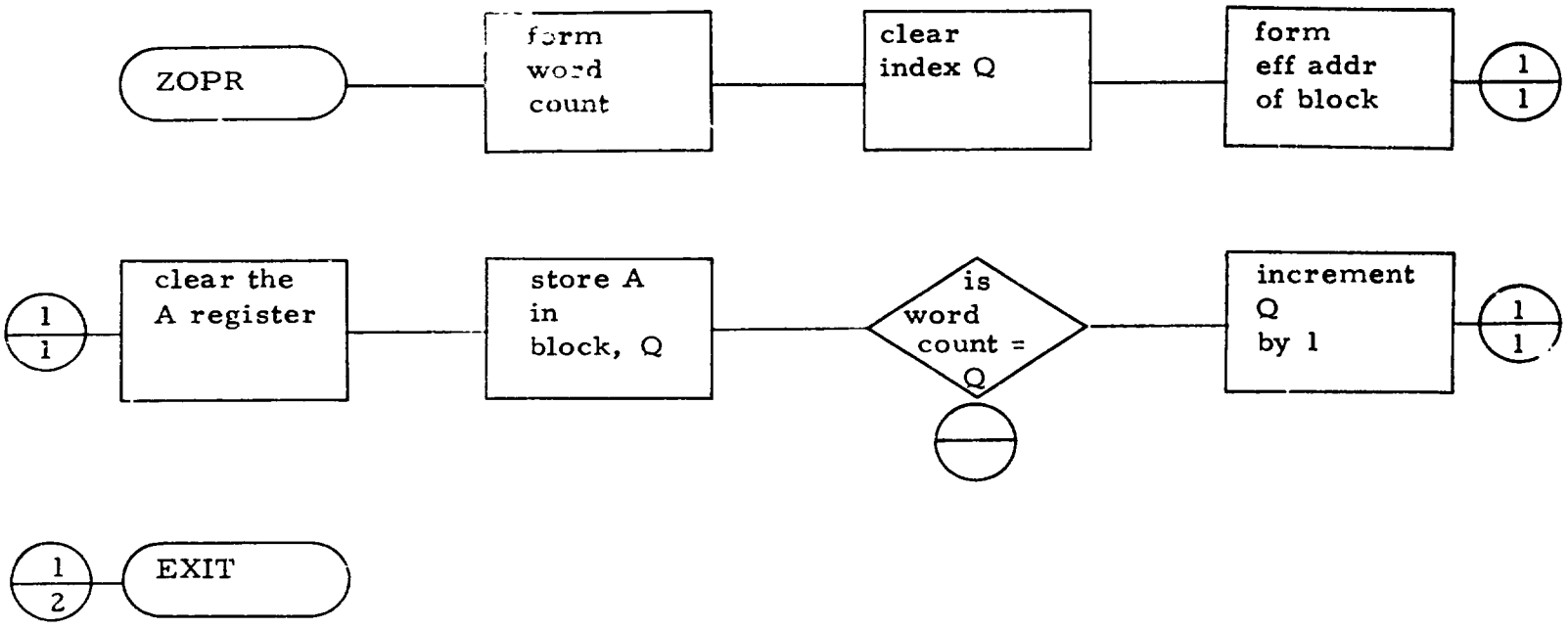


Block Transfer Exchange

Figure 10-12

10.14 ZERO-FILL BLOCK

1. Program Name: ZOPR - Zero-fill Block
2. Abstract: ZOPR stores a zero in every word of the specified block beginning at block address plus index. The number of words made zero is specified by OPN, the word block count.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-13
6. Program Description: Upon entering this program the OPN count is saved. The index into the block is cleared. The effective address of the specified block is formed and saved. Next, the A register is zeroed out and stored in the specified block address plus index. If the block has not been completely zero-filled, the index is increased by 1 and the zero-fill process is repeated. When the block has been completely zero filled, exit.
7. Inputs: /BLOCK/INDEX/OPN/
8. Outputs: A block, OPN words long, is zero-filled.
9. Call Sequence: RTJ ZOPR



Zero-Fill Block

Figure 10-13

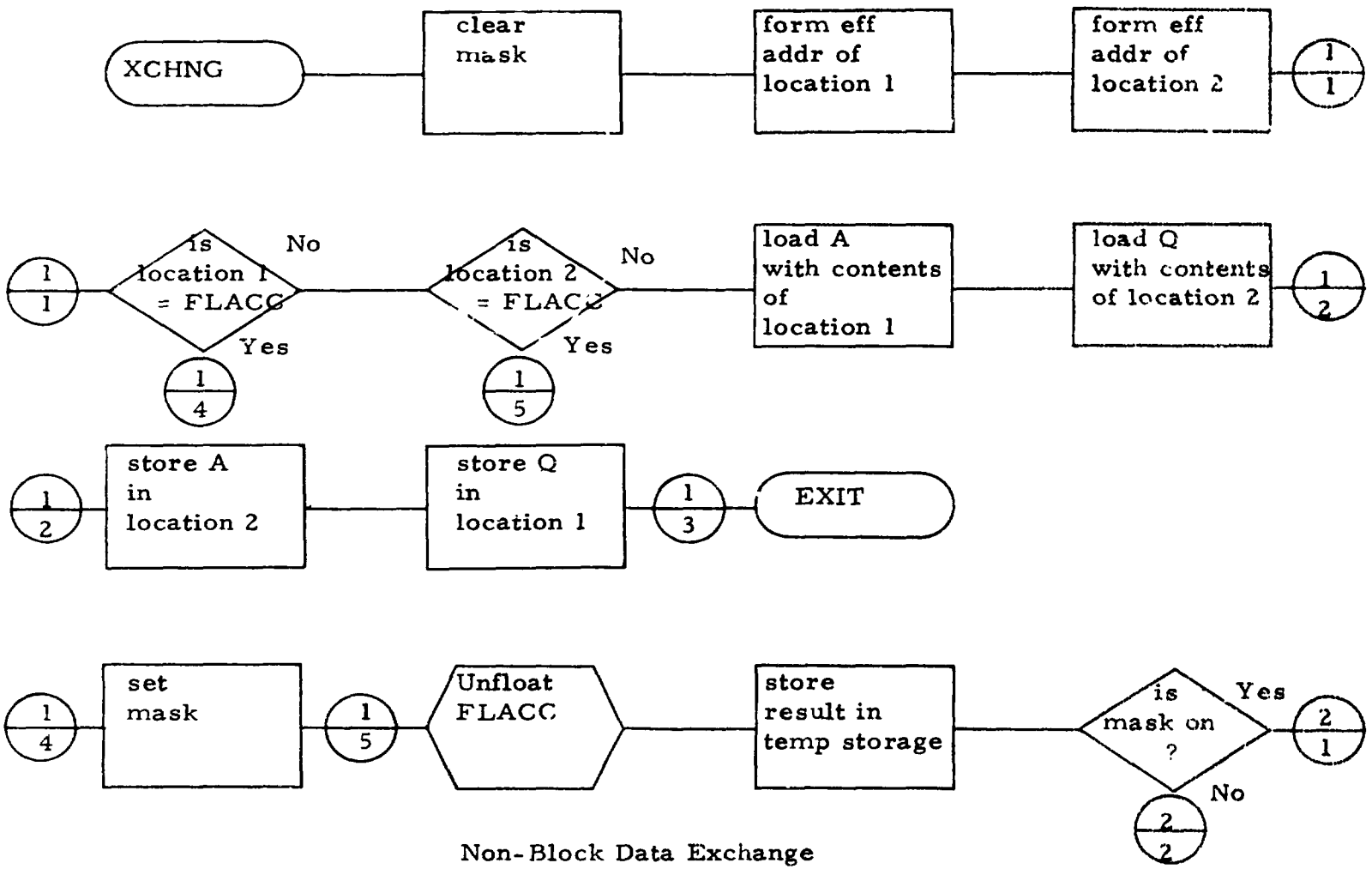
10.15 NON-BLOCK DATA EXCHANGE

1. Program Name: XCHNG - Non-Block Data Exchange
2. Abstract: Non-block data is exchanged between different locations. These may be two fixed point locations or a fixed point and a floating point location. In the latter case, the data is first converted to either fixed or floating point before exchanging.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-14
6. Program Description: On entry into the program the effective addresses of data locations are formed. If both locations are fixed point data a simple exchange (swap) of the data is performed. If location one is a floating point number and location two is a fixed point number, the floated data is unfloated and the fixed data is floated. They are then exchanged (swapped). If location two is a floating point number and location one is a fixed point number, the unfloat and float procedure is performed on these locations and then they are exchanged (swapped). Exit.

   NOTE: The procedure to float a fixed point number and to unfloat (fixed point) a floating point number will be discussed in Section 10.18 and 10.19. The reason to discuss these procedures in another section is that they are common to other

programs and are of such length that they deserve consideration as main programs and not merely subroutines.

7. Inputs: /BLOCK/INDEX/OPN/BLOCK/INDEX/OPN/
8. Outputs: Two data locations are exchanged (swapped)
9. Call Sequence: RTJ XCHNG



Non-Block Data Exchange

Figure 10-14

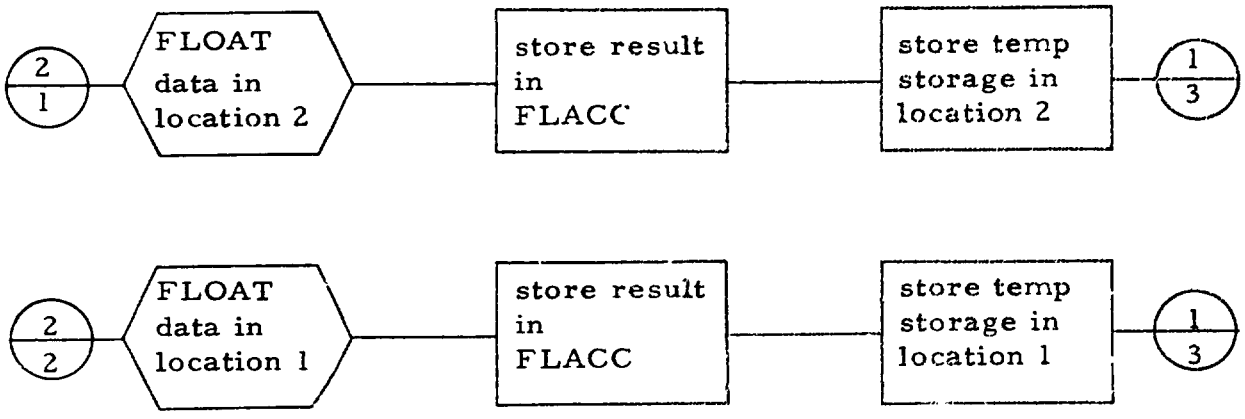


Figure 10-14 (Cont'd)

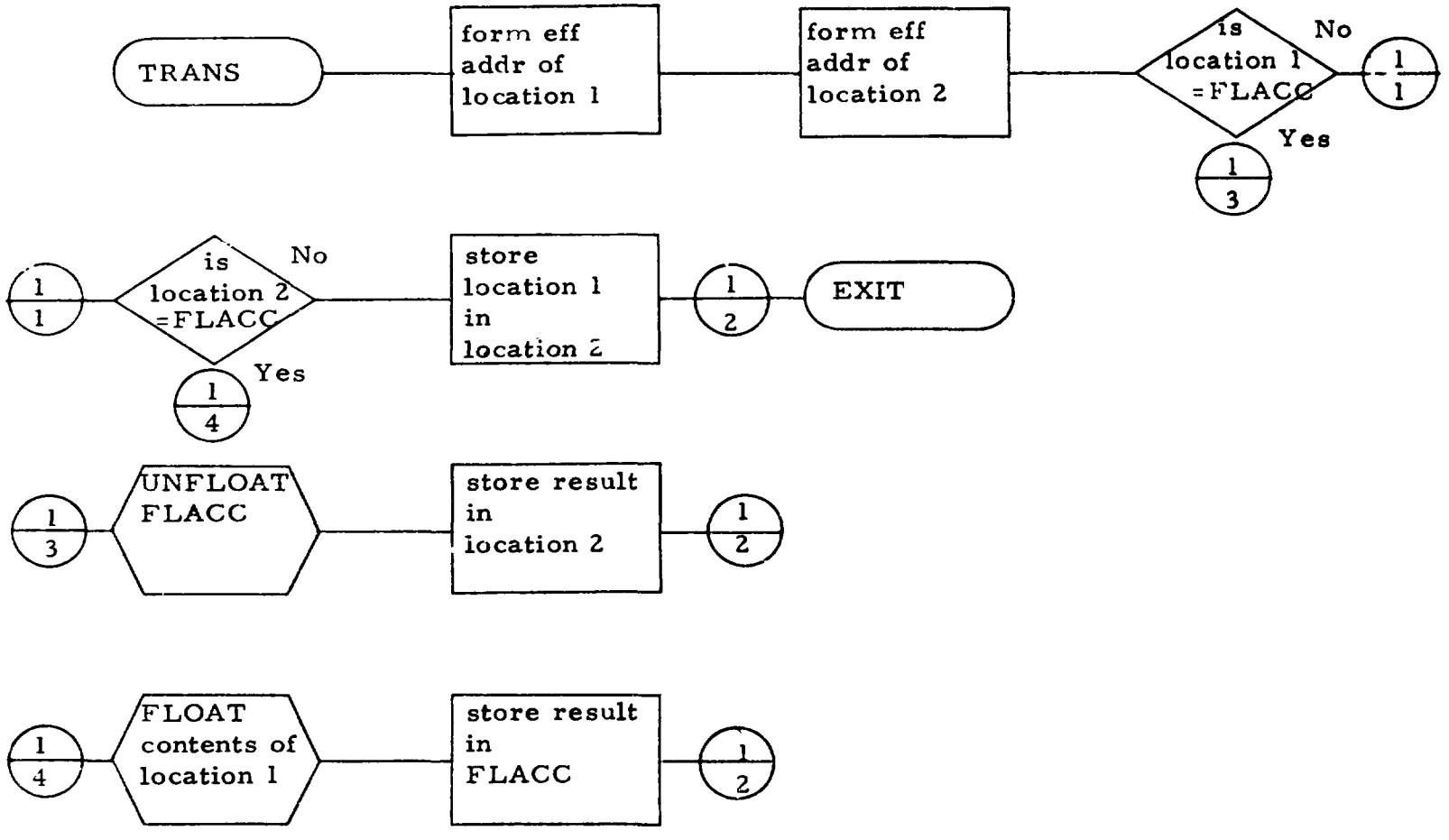
10.16 NON-BLOCK DATA TRANSFER

1. Program Name: TRANS - Non-block Data Transfer
2. Abstract: TRANS transfers data from one location to another. The original data is left unchanged in the originating location. The transfer may be from a fixed point to a floating point location or from a floating point to a fixed point location. In the former case the fixed point is floated and in the latter the floating point is unfloated (fixed point).
3. Operational Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-15
6. Program Description: Upon entry of the program location one is tested against the address of FLACC. If they are the same, FLACC is unfloated and stored in location two. If location one is not the same as the address of FLACC, location two is tested against the address of FLACC. If they are the same, location one is floated and stored in FLACC. If neither location is the same as the address of FLACC, location one is stored in location two.

   NOTE: Refer to the Note in Section 10.15.
7. Inputs: /BLOCK/INDEX/OPN/BLOCK/INDEX/OPN/

8. Outputs: Contents of one location is transferred to another.

9. Call Sequence: RTJ TRANS

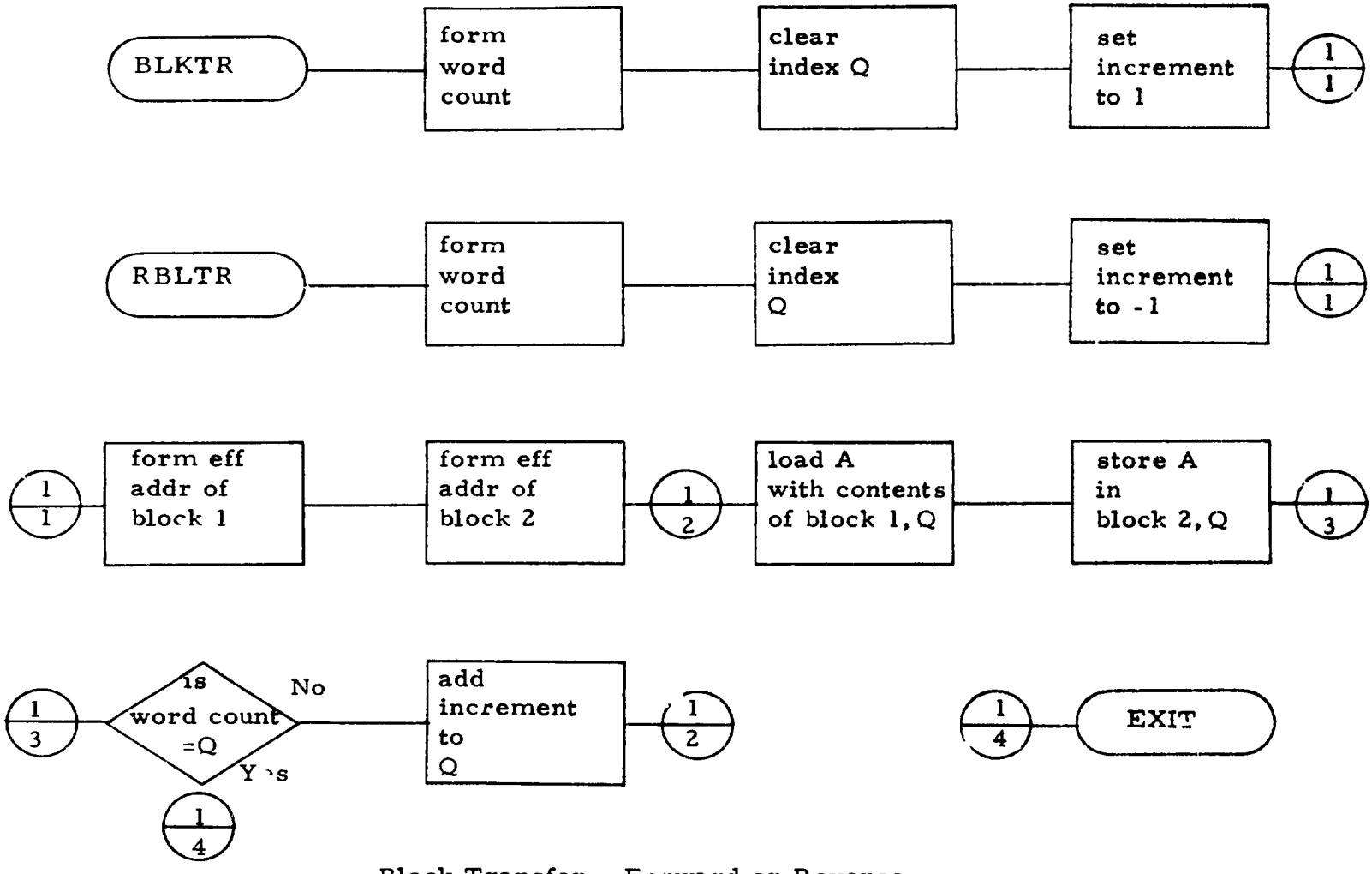


Non-Block Data Transfer

Figure 10-15

10.17 BLOCK TRANSFER

1. Program Names: BLKTR - Block Transfer Forward
   RBLTR - Block Transfer Reverse
2. Abstract: BLKTR transfers a string of data from one block to another leaving the originating block unchanged. RBLTR performs the same function as BLKTR but in a reverse manner.
3. Operational Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-16
6. Program Description: On entry of both programs OPN, the block length, is calculated. If the two blocks are of different length, OPN of the shorter block is chosen as OPN for both. The index Q is cleared. BLKTR sets the increment to 1 and RBLTR sets the increment to -1. Both programs now do a return jump to a subroutine. This subroutine forms the effective addresses of the two blocks. Block address one plus index Q is stored in block address two plus index Q. If OPN words have not been processed add the specified increment (1 or -1) to index Q. The transfer procedure is repeated until OPN words have been processed. Exit.
7. Inputs: /BLOCK/INDEX/OPN/BLOCK/INDEX/OPN/
8. Outputs: Contents of one block is transferred to another.
9. Call Sequence: RTJ BLKTR or
   RTJ RBLTR



Block Transfer - Forward or Reverse

Figure 10-16

10.18 FLOATER

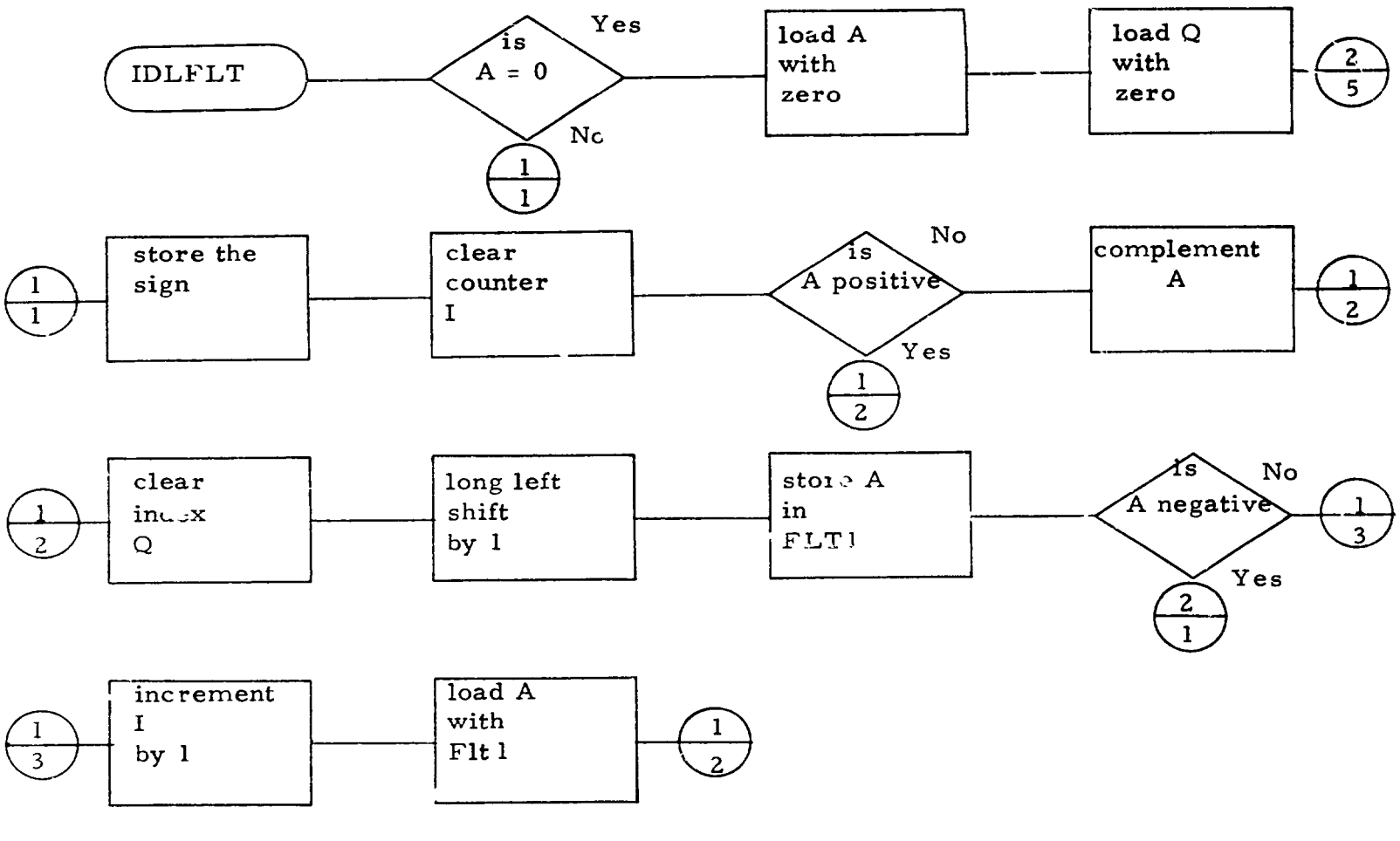
1. Program Name: IDLFLT-Float a fixed point number
2. Abstract: IDLFLT converts a fixed point number entered through the A register into a floating point number. The converted number is left in the A and Q registers with the most significant part in the A and least significant in the Q.
3. Operational Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-17
6. Program Description: The fixed point number is expected to be in the A register. A is checked to see it it is zero. If it is, both the A and Q registers are set to zero and exit occurs. If the A register is not zero, its sign is saved in temporary storage. The sign is tested and if it is negative, "A" is complemented, otherwise, nothing is done to it. The next step is to find the first significant bit of the fixed point number. This is done by shifting the word left one bit at a time and testing the left-most bit. When the first "1" bit is encountered, the word is saved in FLT1 and the shifting process is stopped. The number of times the word was shifted one bit is now recorded. The exponent of the floating point number is formed by first subtracting this recorded counter from 15 and then adding it to 80

This value is now placed in bits 7-14 by shifting left by 7 and saved for later. Remembering that FLT1 contains the significant bits of the number, it is loaded into the A register. The Q register is cleared and a long left shift by 7 is performed. This places the first 7 bits of the number in position 0-6 of the Q register. The A register contains the least significant part of the floating point word and is stored in FLT2. Now, an "exclusive or" of the A register and the saved exponent is executed. This places the exponent in bits 7-14 and the most significant part of the number in bits 0-6. This is saved in FLT1. Finally, the sign that was saved is tested. If it is negative both FLT1 and FLT2 are complemented. If the sign is positive FLT1 and FLT2 are not changed. As a final step, the A register is loaded with FLT1 and the Q register with FLT2. Exit.

7. Inputs: The A register must contain the fixed point number.

8. Outputs: The A and Q registers are returned with the floating point number, the least significant part being in the Q.

9. Call Sequence:
```
LDA FXACC
RTJ IDLFLT
STA FLACC
STQ FLACC+1
```



Float Fixed Point

Figure 10-17

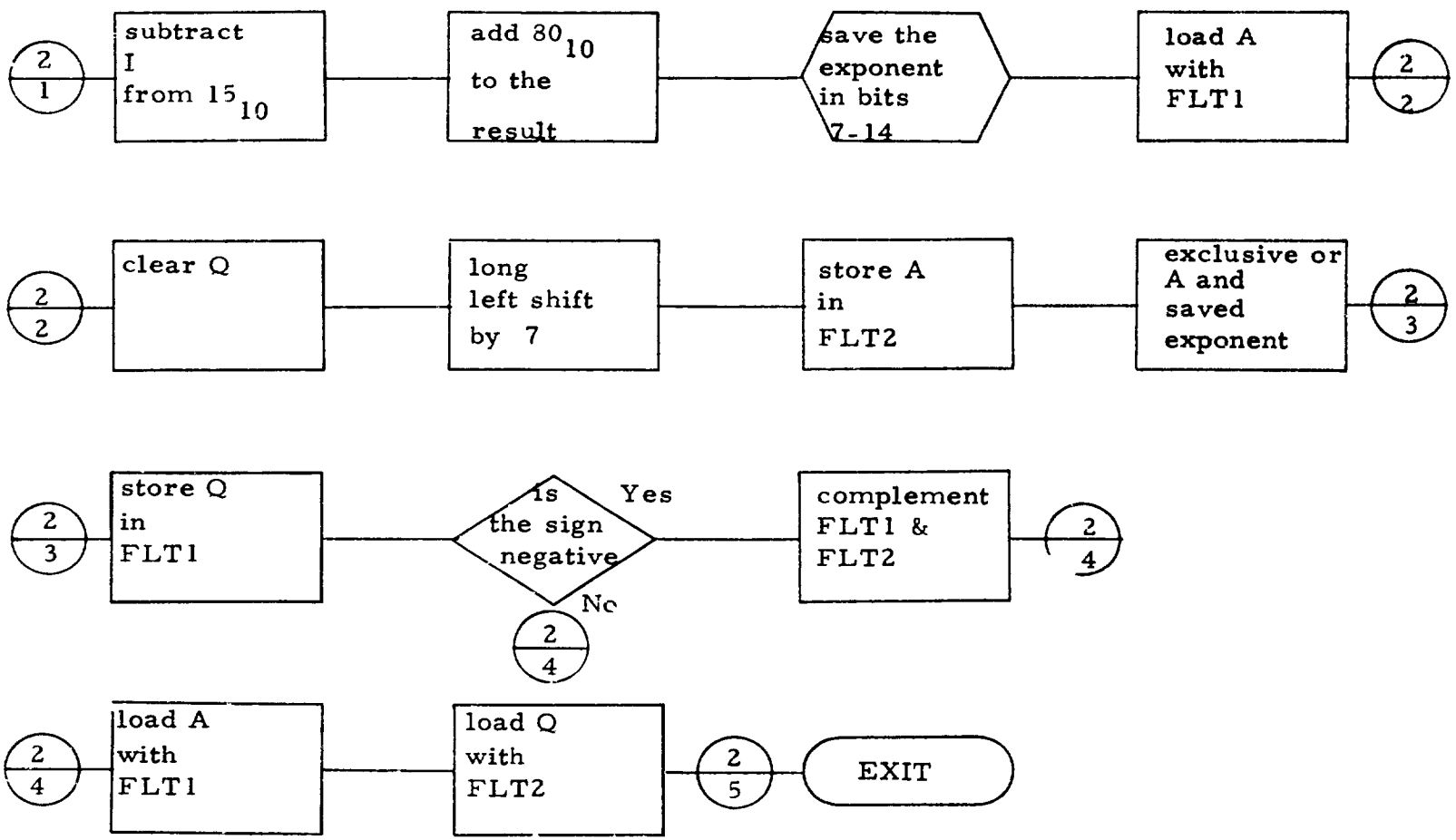


Figure 10-17 (Cont'd)

10.19 UNFLOATER

1. Program Name: IDLUNF - Unfloat a floating point number
2. Abstract: IDLUNF converts a floating point number from the A and Q registers (A contains the most significant part and Q the least significant) into a fixed point number. The result is returned in the A register.
3. Operational Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-18
6. Program Description: On entry to the program the A and Q registers are stored in FLT1 and FLT2 respectively. The sign of A is stored for later use. If the sign is negative both FLT1 and FLT2 are complemented. If the sign is positive FLT1 and FLT2 are unchanged. The exponent of the floating point number is extracted from bits 7-14 of FLT1. This exponent value is tested. If it is zero or negative the A register is loaded with zero and exit occurs. If the exponent is positive, it is decreased by 1 and set into counter I. If the counter is greater than 15 the fixed point overflow FXOVFL is set, the A register is loaded with $7FFF_{16}$ and exit occurs. If the counter is equal or less than 15, FLT1 and FLT2 are loaded into the Q and A registers respectively and a long right shift by 7 is executed. This results in placing the

co-efficient of the floating point number in A. Store A in temporary storage.

Now, the counter I is subtracted from 16 and stored in the Q register as a counter for the procedure below.

In order to produce the fixed point number, bits 15 through x (x=15-Q+1) of the stored co-efficient are used in the following arithmetic formula:

$FXTPI = B_{15}*2^{p} + B_{14}* 2^{p-1} +.....+B_{x}*2^{o}$,

where B_n = bit n of the co-efficient

p = Q-1 (number of bits -1)

x = 15-Q+1 (last processed bit)

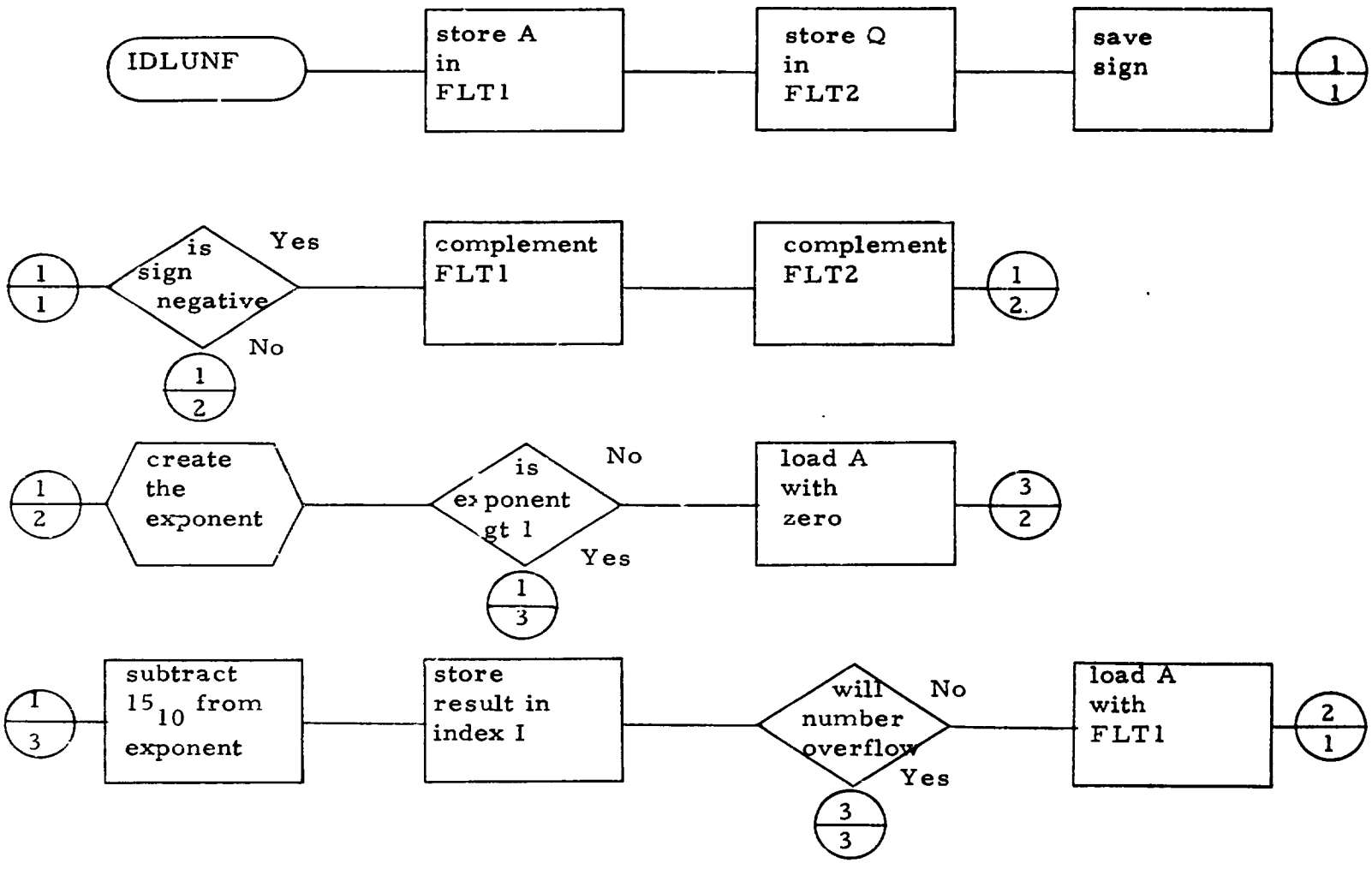
The result of the formula is the converted fixed point number. This is loaded into the A register. The saved sign of the floating point number is tested. The A register is complemented if it is negative or left unchanged if it is positive. Exit.

7. Inputs: The A and Q register must contain the floating point number. The most significant part is in A and the least significant in Q.

8. Outputs: The A register is returned with the fixed point number

9. Call Sequence:

```
LDA  FLACC
LDQ  FLACC+1
RTJ  IDLUNF
STA  FXACC
```



Unfloat Floating Point Number
Figure 10-18

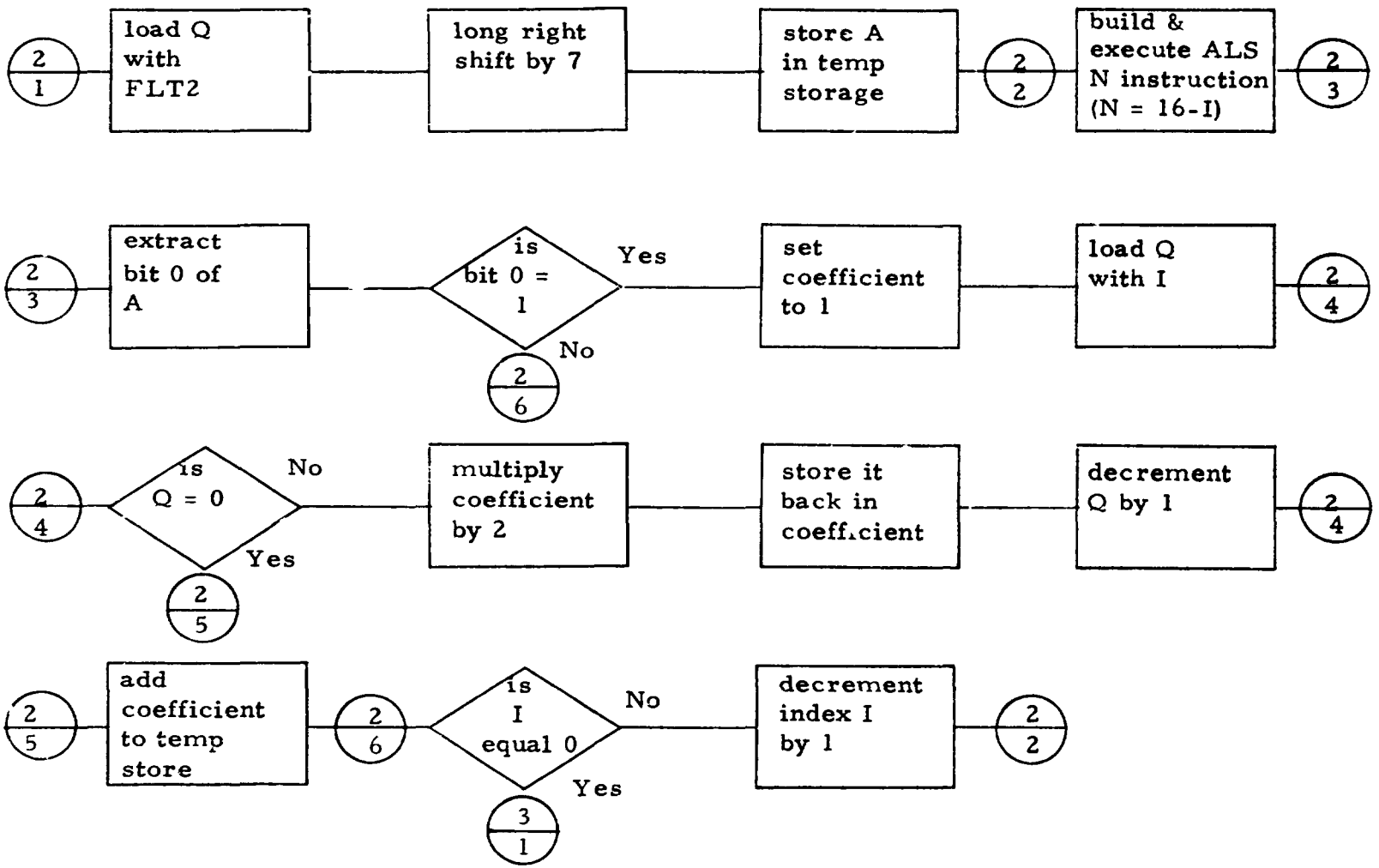


Figure 10-18 (Cont'd)

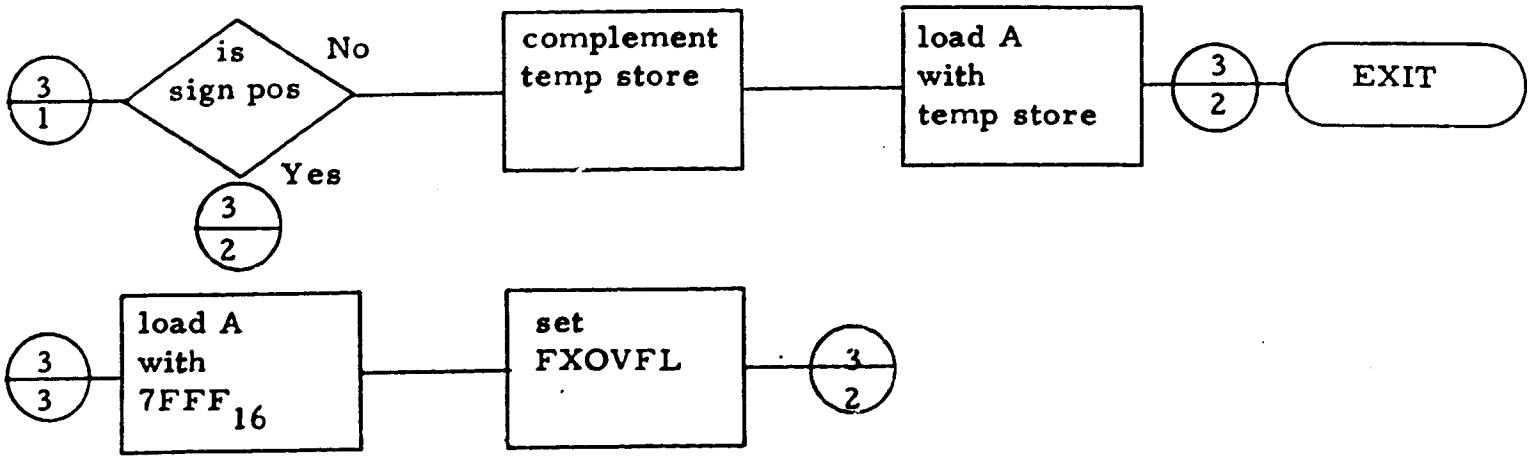
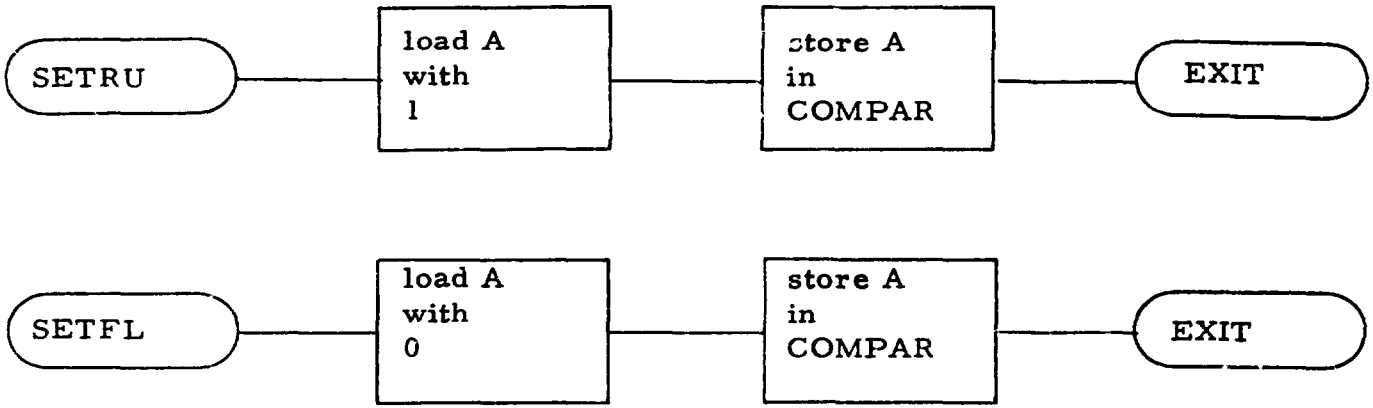


Figure 10-18 (Cont'd)

10.20 SET COMPARE INDICATOR

1. Program Names: SETRU - Set comparator true
   SETFL - Set comparator false
2. Abstract: SETRU sets the compare indicator COMPAR true (1).
   SETFL sets the compare indicator COMPAR false (0).
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-19
6. Program Description: SETRU loads the A register with 1 and stores it in COMPAR. Exit.
   SETFL loads the A register with 0 and stores it in COMPAR. Exit.
7. Inputs: None
8. Outputs: COMPAR is set true or COMPAR is set false.
9. Call Sequence: RTJ SETRU or
   RTJ SETFL

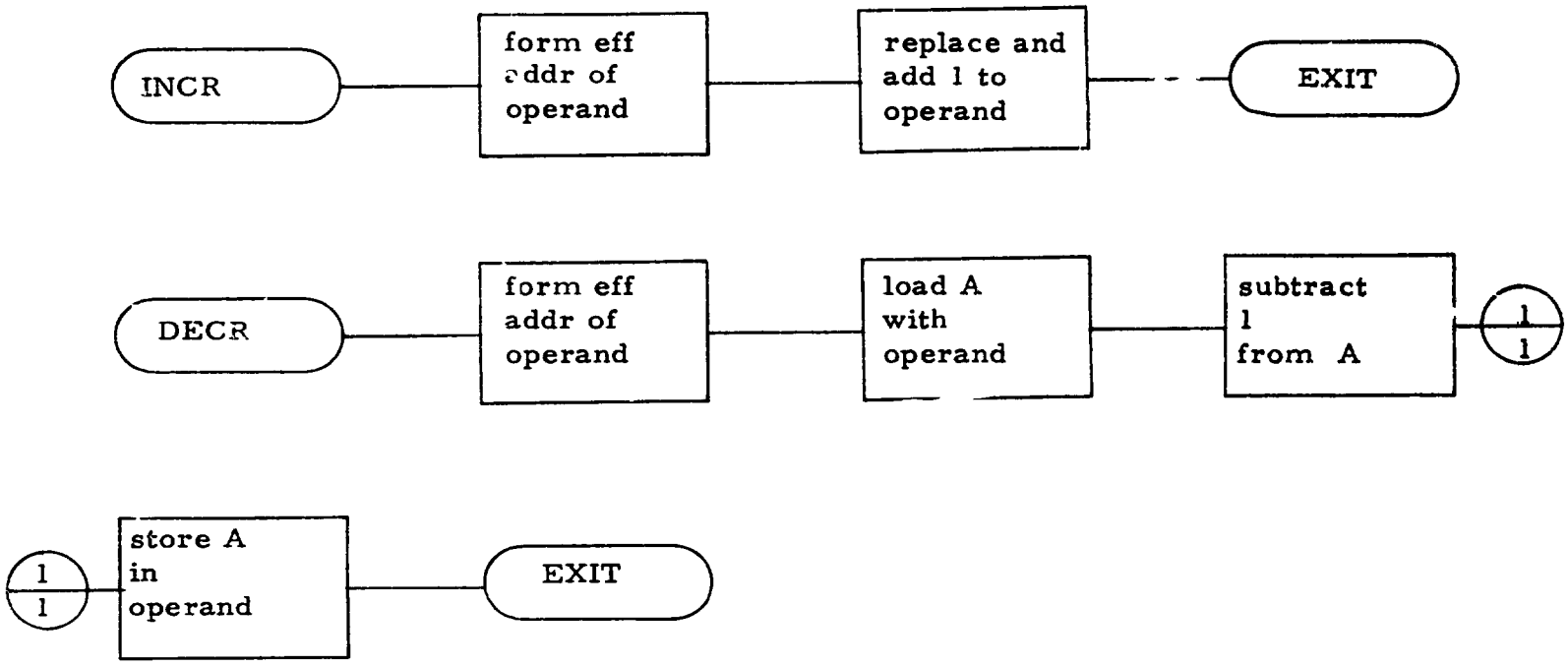


Set COMPAR Indicator

Figure 10-19

10.21 INCREMENT-DECREMENT OPERAND

1. Program Names: INCR - Increment Operand
   DECR - Decrement Operand
2. Abstract: INCR increments a given operand by 1.
   DECR decrements a given operand by 1.
3. Operational Requirements: CDC 1700
4. Language: CDC 1700
5. Functional Flow Charts: See Figure 10-20
6. Program Description: Both programs form the effective address of the operand. INCR replaces and adds one to the operand. DECR loads the A register with the operand and subtracts 1 from it. It then stores A back into the operand. Exit.
7. Inputs: /BLOCK/INDEX/OPN/
8. Outputs: The operand is either incremented or decremented by 1
9. Call Sequence: RTJ INCR or
   RTJ DECR



Increment or Decrement Operand

Figure 10-20

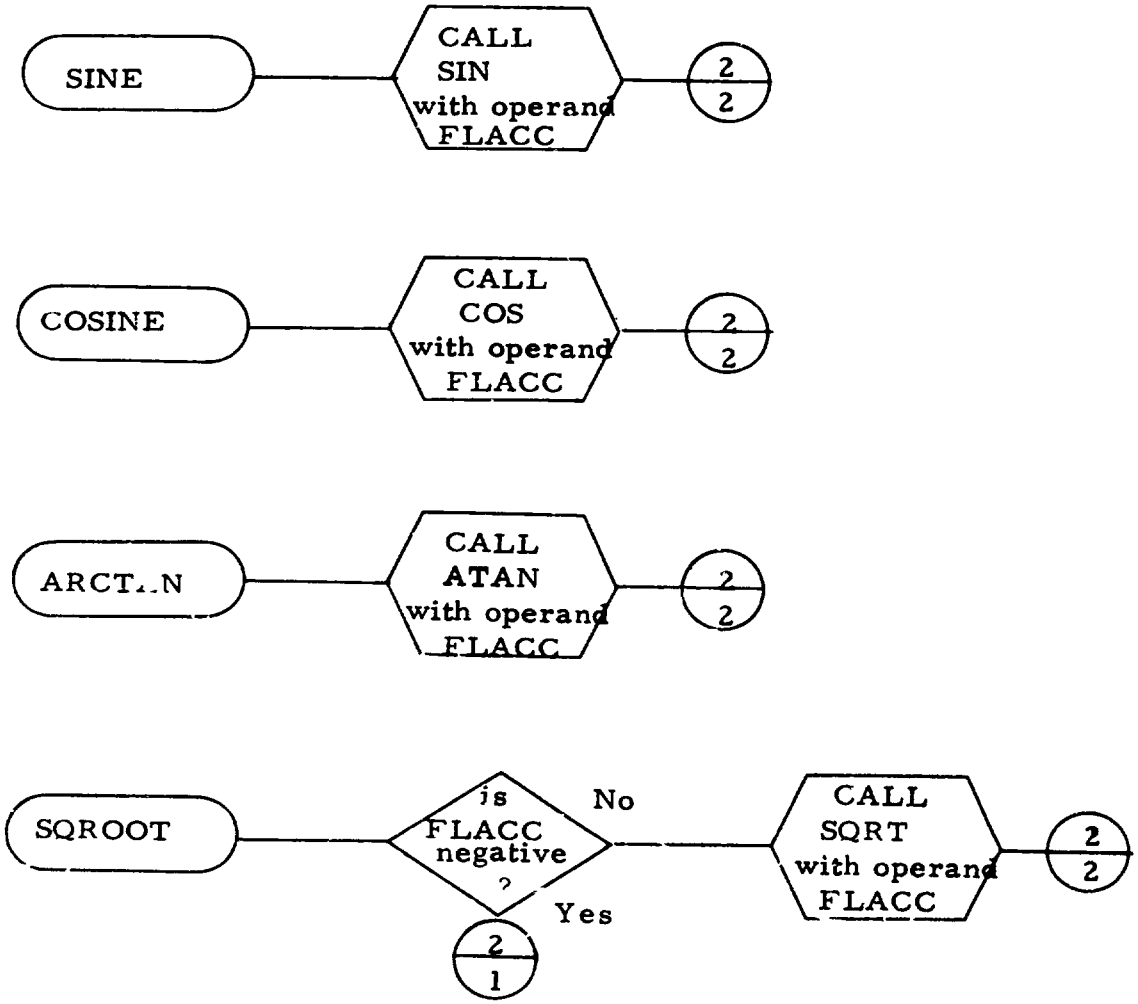
10.22 TRIGONOMETRIC FUNCTIONS

1. Program Names: SINE - sine of FLACC
   COSINE - cosine of FLACC
   ARCTAN - arctangent of FLACC
   SQROOT - square root of FLACC
   LOGAR - logarithm of FLACC
   ANTLOG - antilog of FLACC
2. Abstract: FLACC is used as the operand for producing the result of the desired trigonometric function. This result is returned to FLACC, therefore, replacing the original operand.
   NOTE: The log and antilog functions are to the base 10.
3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-21
6. Program Description: Each program has its own entry point. Their basic procedure is to call the CDC 1700 system trigonometric function with FLACC as the operand. After the result is calculated the CDC 1700 system floating point package FLOT is called so that this result can be stored back in FLACC. While in the FLOT package a check for overflow is made. If overflow occurs, the floating point overflow indicator FLOVFL is set. Otherwise, exit.
   NOTE: The function LOGAR does not perform the operation if the operand is equal or less than zero. SQROOT also does same if operand is negative. FLOVFL is set for both conditions.

7. Inputs: None

8. Outputs: The result of the function is returned in FLACC. FLOVFL is set on overflow.

9. Call Sequence:
```
RTJ  SINE    or
RTJ  COSINE  or
RTJ  ARCTAN  or
RTJ  SQROOT  or
RTJ  LOGAR   or
RTJ  ANTLOG
```



Trigonometric Functions

Figure 10-21

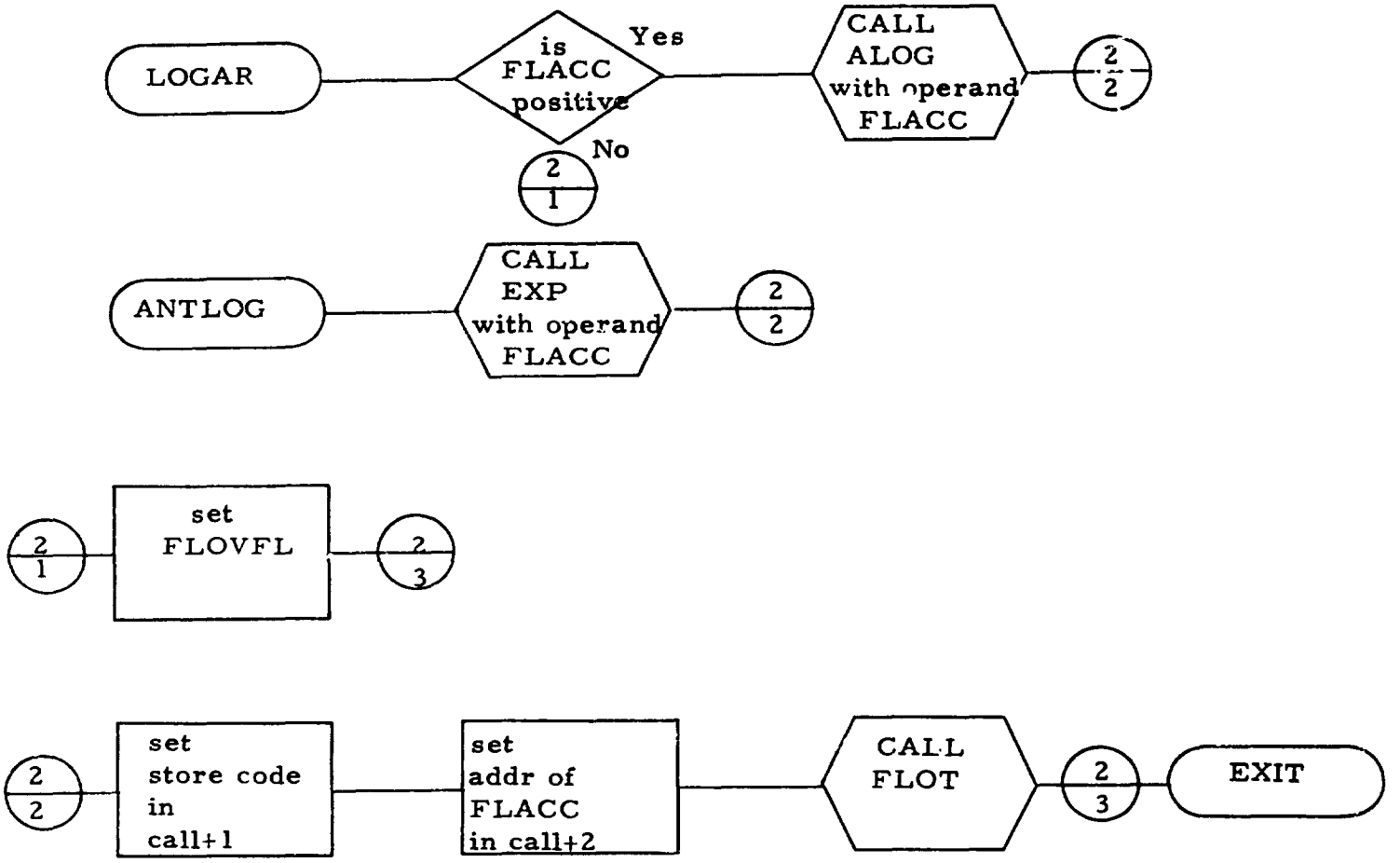


Figure 10-21 (Cont'd)

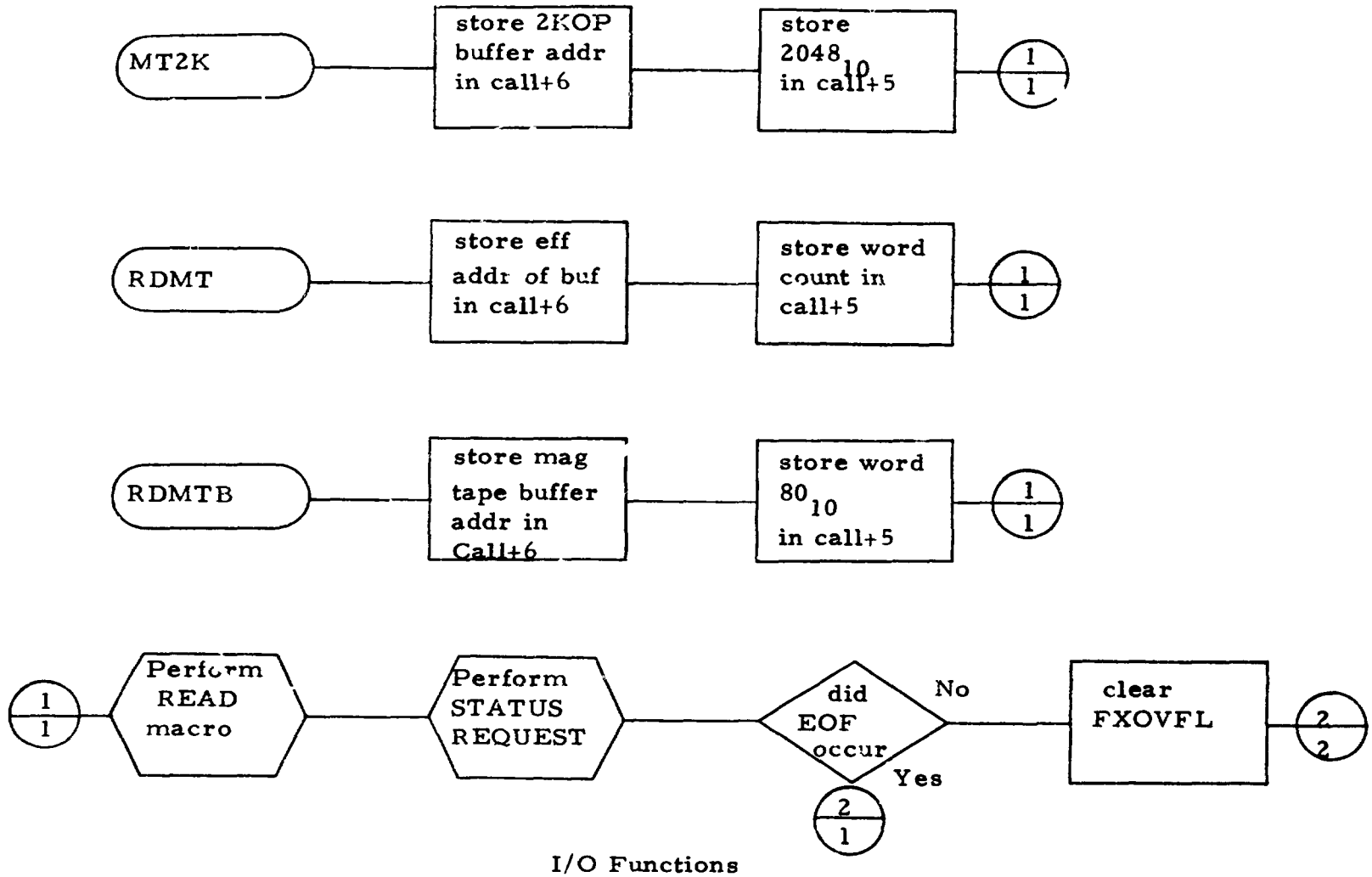
10.23 INPUT/OUTPUT

1. Program Names: MT2K - Read magtape into 2K buffer
RDMT - Read magtape into any buffer
RDMTB - Read magtape into magtape buffer
A2KMT - Write to magtape from 2K buffer
WRMT - Write to magtape from any buffer
WRMTB - Write to magtape from magtape buffer
RWNMT - Rewind magtape
BKSP - Backspace magtape
WREOF - Write end-of-file mark
RDEOF - Read to end-of-file
WRLP - Write to line printer from any buffer

2. Abstract: Data to or from three types of buffers (2K-OP, MTB, BLOCK) can be transferred to or from magtape. MT2K, RDMT, RDMTB, A2KMT, WRMT, and WRMTB will perform these functions. RWNMT, BKSP, WREOF and RDEOF will perform tape motions as described in the program names above.
WRLP takes data from a buffer and outputs it to the line printer.

3. Operating Requirements: CDC 1700
4. Language: CDC 1700 Assembler
5. Functional Flow Charts: See Figure 10-22
6. Program Description: Each routine has its own entry point. The buffer address, word count and proper request code are built into call +1 through call +6 for the call to the CDC 1700 I/O driver. These variables can be found in the CDC 1700 operating systems manual. A jump to the built code is made. This causes the requested magtape I/O to be performed. When this is accomplished, exit.
7. Inputs: /BLOCK/INDEX/OPN/ (optional)
8. Outputs: Data transfer between a buffer and magtape and a repertoire of tape motions.
9. Call Sequence:

```
RTJ  MT2K or
RTJ  RDMT or
RTJ  RDMTB or
RTJ  A2KMT or
RTJ  WRMT or
RTJ  WRMTB or
RTJ  RWNMT or
RTJ  BKSP or
RTJ  WREOF or
RTJ  RDEOF or
RTJ  WRLP
```



I/O Functions
Figure 10-22

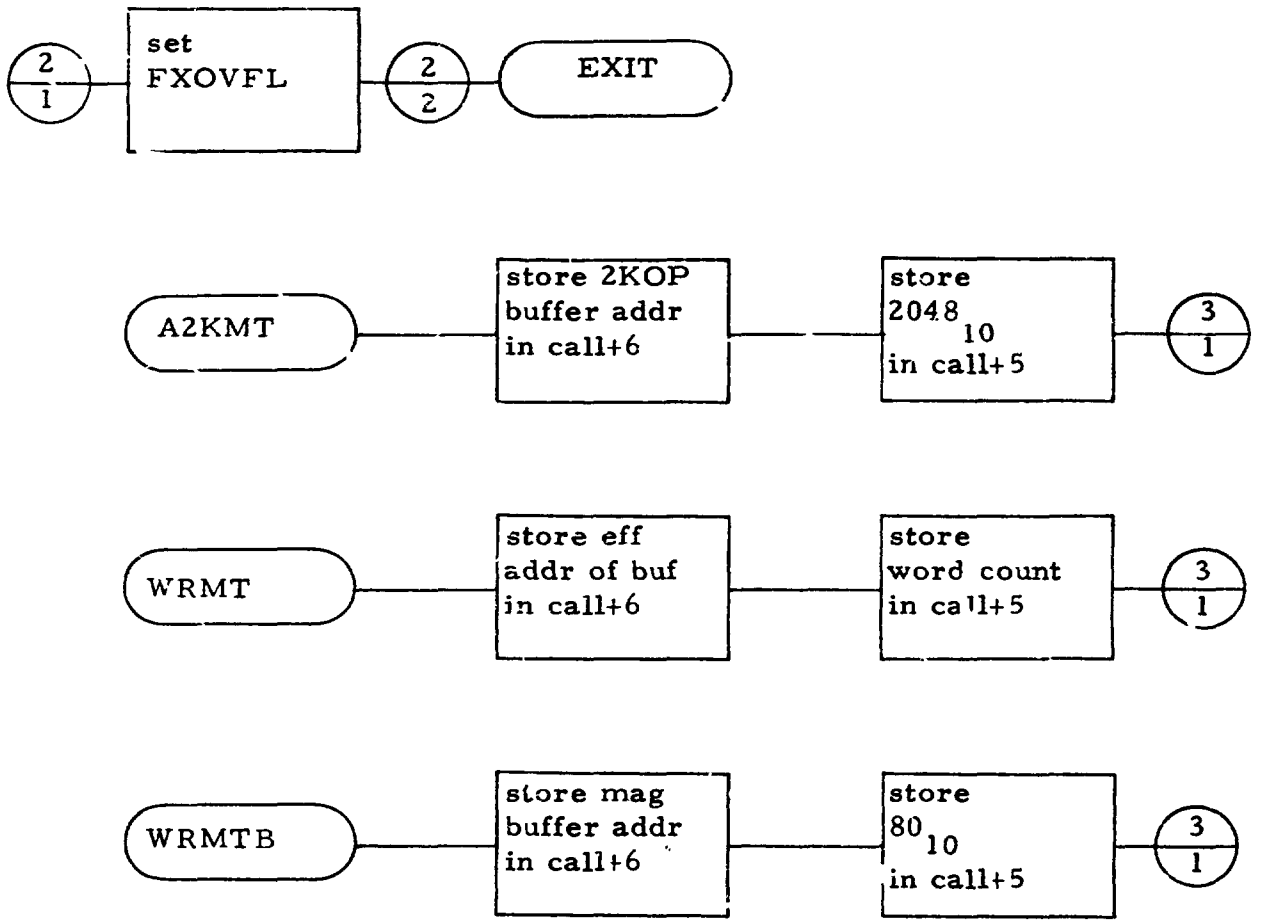


Figure 10-22 (Cont'd)

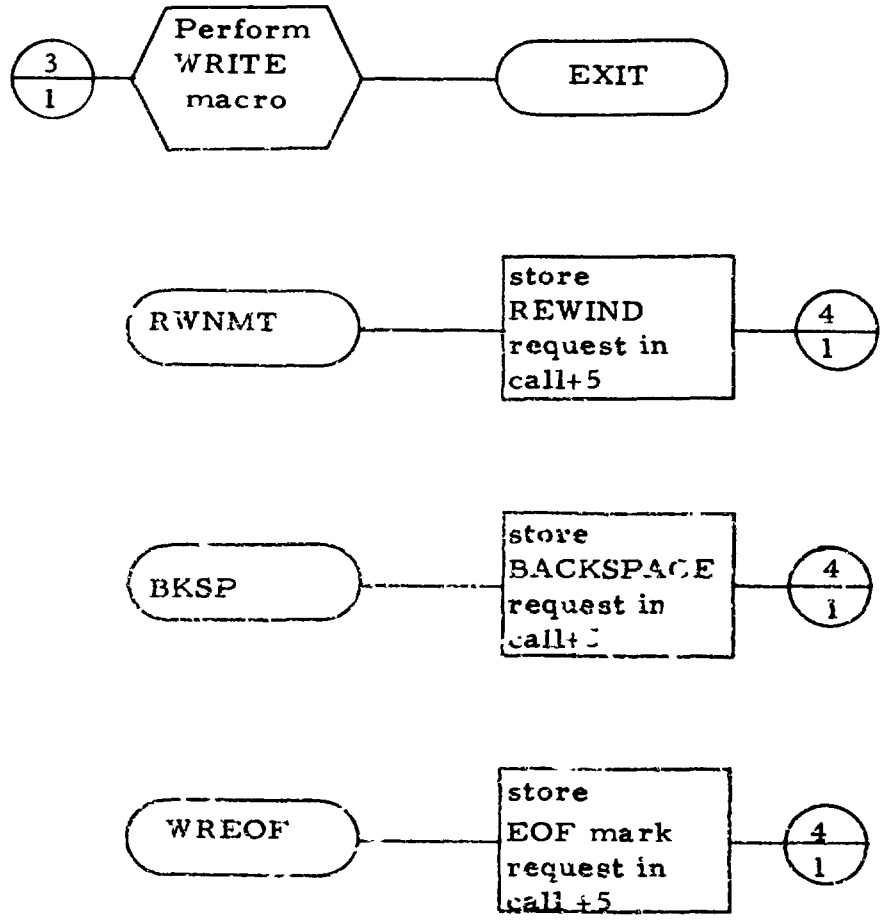


Figure 10-22 (Cont'd)

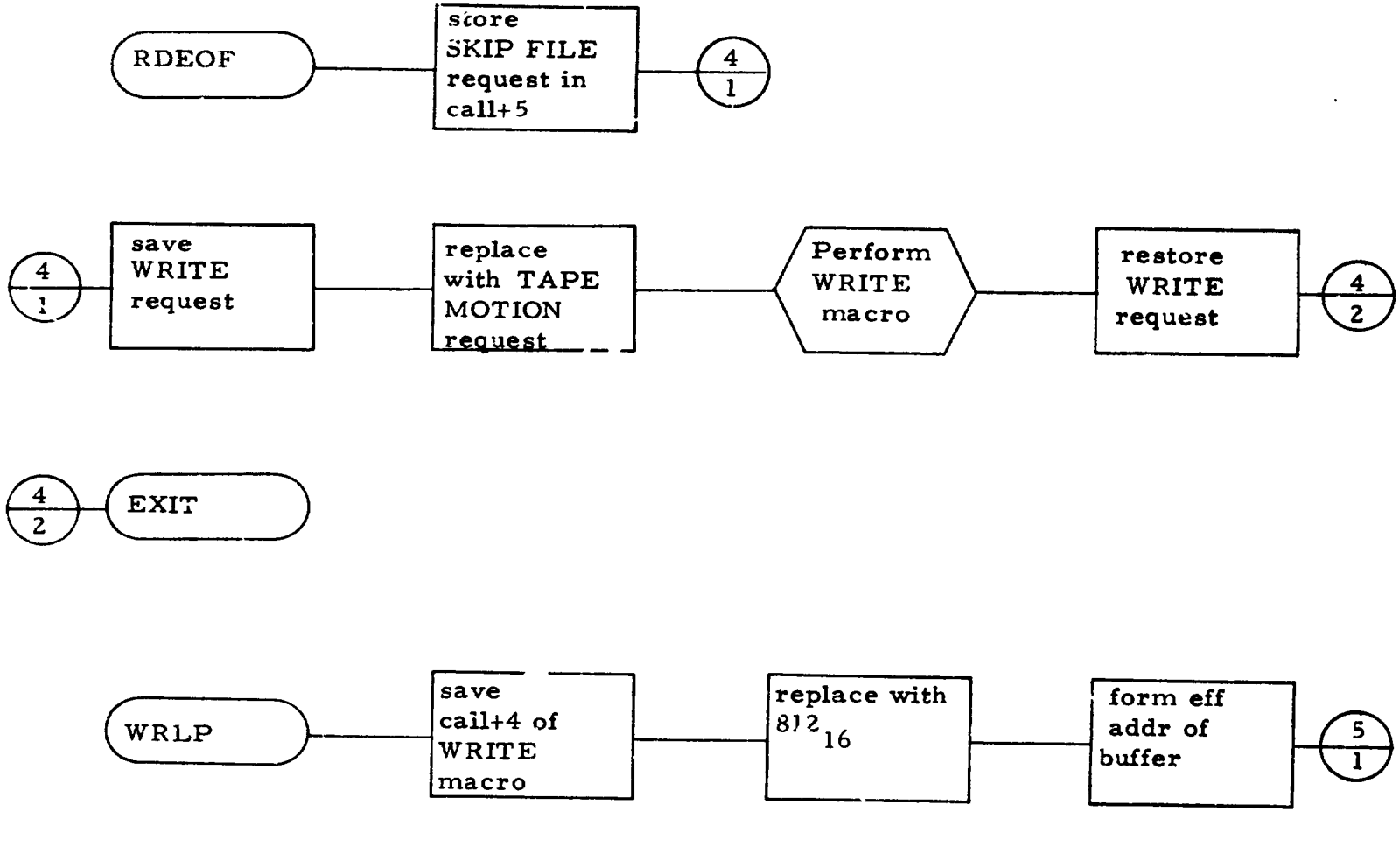


Figure 10-22 (Cont'd)

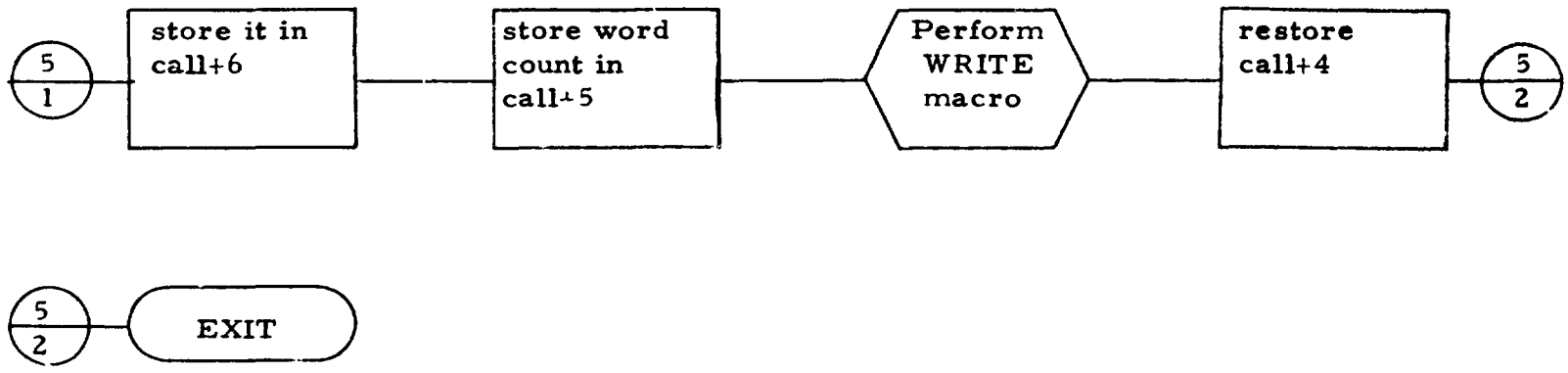


Figure 10-22 (Cont'd)

SECTION 11

CONCLUSIONS

The current implementation of IDL has lead to several conclusions about the concept. These conclusions are based on deficiencies in IDL discovered during the implementation process. Although these deficiencies do not obviate the IDL concept of an interactive programming language they show that the technical approach taken in implementing the concept was not based on the user's requirements.

The basic conclusions are as follows:

- o IDL is completely programmer oriented. It can only be used by a skilled programmer.
- o Because it is based on the simulation of a psuedo computer a large amount of the available storage must be dedicated to the simulation routines.
- o IDL commands are based on the depression of keys on the BR 90 Variable Function Keyboard. The distribution of the command set across 7 overlays requires that overlays must be physically changed after almost every key depression.
- o The totality of all IDL Programs, Subroutines and Labels is restricted to the keys available on the BR 90 overlay set. Since this is so only 1920 Identifiers are available for the total IDL implementation.
- o The current IDL implementation provides no method of debugging an IDL program.

The above are major deficiencies in the IDL implementation and essentially negate the interactive concept. Any further effort on the current design would be wasted effort unless the problem areas are resolved.

APPENDIX A

Key Descriptions

1. ASSEMBLER FUNCTIONS:

DEFIN 0513 — The DEFIN key is used to indicate the beginning of a console program. The following key will be the "Title" (or name) of the program being written. This title is the octal number by which the program will alway ; be referenced.

LNPRT 0416 — The Line print function causes the program or operand of the next key(s) entered to be printed, in octal key signature format, on the line printer.

STOP 0507 — This key causes the assembled program to be stored on the Disk File and the permanent Key Identification Table upda:ed.

START 0506 — This key is accepted by the basic system but causes no action to occur.

ASIGN 0525 — The Assign Key is used to specify a data storage area (sometimes called an array or an operand.) The next key entered will be the "Title" of the operand that is being specified. The next entry must be the dimension of the operand. This is done by entering the number keys on overlay 05. The dimension will be considered as a decimal number and the DEC or OCT keys must not be used. Following entry of the number key(s) for the dimension the Field "Delimiter" Key (/0536) must be entered. A feature is provided whereby the contents of the storage area may be specified. This is accomplished just as one would insert constants into a program (using the OCT and DEC keys). If all words of an operand are not specified, the remaining words will be set to the

value of the last specified word. If no contents are specified, then zero is stored throughout the operand. Finally, the "End" key, 0514, must be entered to terminate the assignment of the operand. It should be remembered that the cells of the data storage are numbered 0 through N-1.

END 0514 — This key is used to terminate the assembly of a program being defined or to terminate the assignment of an operand. (An error condition will result if the assembler is not in the "Defin" (0513) or "Asign" (0525) mode.)

UNDEF 0526 — This key is used to undefine a key which has been previously defined or assigned. The next key entered is the key which will be undefined. Note that no key on overlays 00 through 06 may be undefined. Also note that when the undefine key is entered, some undefinable key must subsequently be entered. It is acceptable to undefine a key which is already undefined.

EXECU 0532 — This key when entered places the system in the "Execute Idle" Mode. The next key entered causes the system to execute the program defined for that key. This program is a "main" program (by virtue of being called for while in "Execute Idle" mode.) When the "End" of a main program is encountered the system returns to the "Execute Idle" mode after emptying core. (Note that "Abort" or an interpretive error will also cause a return to the "Execute Idle" mode.) The execute mode is

terminated whenever an "Assembler" function key is entered while the system is in the "Execute Idle" mode. Thus, it is not necessary (although acceptable) to enter the Execute Key, 0532, between executions of main programs as long as an assembler function has not been called for.

Label 0521 — Label follows provides a method for determining the destination of a jump. A label must appear immediately after 0521. A label may be any key on any overlay except the increment and decrement keys. The key used as a label may have any other system meaning or may be undefined. This label is the destination referred to by a jump. (? it must conform exactly to the key referenced in the jump, i.e. overlays must match.) The destination of a jump referring to the label will be the instruction immediately following the label. Keys used as labels are only effective in one program (as delimited by its DEFIN and END)--that is, a jump from one program into another is disallowed.

2. CONSTANTS:

Constants may be assembled into a program or operand by using the following keys:

DECFL 0503	Assemble a floating point number.
DEC 0504	Assemble a decimal fixed point number.
OCT 0505	Assemble an octal fixed point number. One of the above keys must precede each constant entered into the system.
NEG 0512	Make number being assembled negative (twos complement). This key may be entered any time after the designator and before the delimiter.
DECPT 0511	Decimal point for floating point number.
INFIN 0534	Make fixed point number largest possible; i.e. $7FFF_{16}$.
DEL 0536	Delimit number being assembled; i.e., last key entered for a constant.

0.... 0535
2.... 0530
3.... 0527
4.... 0524
5.... 0523
6.... 0522
7.... 0517
8.... 0516
9.... 0515

} Number keys

} May not be used after OCT 0505

Example: To enter -79.65 use:

503/512/517/511/522/523/536/

FL - 7 . 6 5 /

3. REGISTERS AND BUFFERS.

FXACC	0614	Fixed point accumulator. (One work register.)
FLACC	0613	Floating point accumulator. (Effectively, a three word register.)
X1REG	0117	
X2REG	0116	
X3REG	0115	
X4REG	0112	
X5REG	0111	Index Registers. (One word registers.)
X6REG	0111	
X7REG	0105	
X8REG	0104	
X9REG	0103	
CURX	0620	Cursor X register. The current X coordinate of the display cursor is copied into this register on each occurrence of the "Read Cursor" (key 0633) operation.
CURY	0621	Cursor Y register. The current Y coordinate of the display cursor is copied into this register on each occurrence of the "Read Cursor" (key 0633) operation.
LGR	0622	Light gun register. The display memory address of the currently "Light Gunned" element on the display is copied into this register on each occurrence of the "Read Cursor" (key 0633) operation. (If the BR-90 does not have an active light gunned element, then zero is copied into this register. If the BR-85 does not have an active light gunned element, the address of the previous light gunned element is copied into this register.)

LTR 0305 Light Register. This is a four-word register that controls the program key (variable function key) lights. Left keyboard 1-17 (octal) are controlled by bits 1-15 of the first word, Lights 20-36 (octal) by bits 1-15 of the second word. Right keyboard lights 1-17 (octal) are controlled by bits 1-15 of the third word, lights 20-36 (octal) by bits 1-15 of the fourth word. Bit 1 of the words are least significant and correspond to lights 1 and 20 (octal) a 1 in a bit position means light on while a 0 indicates light off. The setting of this four word buffer does not effect the actual lights until a "set lights" operation (Key 0306) is performed. Furthermore, the "I/O Buffer Controller" must be informed to stop automatic control of keyboard lights to make this operation meaningful.

A2KOP 0423 This is a 2048 word operand. It is used similarly to any other operand and has the further capability of direct communication to or from a display console (see keys 0430, 0431, 0432 and 0433) or magnetic tape record (see keys 0034 and 0136). It is also used as the data source fro the "Assemble 2K" and "Alternate Assemble 2K" functions (see keys 0427 and 0331).

DMB 0616 Display, Module Buffer. This 80 word buffer is used to communicate with the display console memory. The last two cells determine the memory addresses to be used in the display console. The last cell (79th or 117 octal) contains the beginning display address effected and the next to last cell (78th or 116 octal) contains the

		ending display address affected. (Also see DMFR and DMTO keys 0316 and 0317).
MTB	0617	Magnetic Tape Buffer. This 80 word buffer is used for communication with all I/O devices except the display consoles, plotter and clock.

4. OPERATIONS:

An understanding of data formats and data source types is necessary to properly utilize the non-ambiguous syntax of the basic system operations. Abbreviations used later are explained here.

FXOP — Fixed Point Operand. This is a single 15 bit word. (i.e., OCT 1/or FXACC or X1 or XX...,03). It may be contained within a block of data, but must be specifically identified within the block (i.e., DMB, X1 or A2KOP/OCT5/).

FLOP — Floating Point Operand. This is a set of 3 sequential 15 bit words. They may be contained within a block of data, but the first word of the set must be specifically identified within the block.

INDEX — A single 15 bit word used to specify the element within a block that defines a "FXOP" or "FLOP". An index may be the fixed point accumulator, any one of the nine index registers, any one of the 28 extra index registers or a constant.

BLOCK — A group of 15 bit sequential words (at least two or more) assigned as an operand (see Assign) or one of the basic buffers;A2KOP, MTB, DMB, XXBLK, LTR.

OPN — A 15 bit word used to specify the absolute maximum length of a block operation. Note that the block may be shorter than OPN and in that case the operation would terminate at the end of the block (before OPN iterations had occurred). OPN may be the fixed point accumulator, any one of the nine index registers, any one of the 28 extra index registers or a constant. Any time OPN is

specified in a syntax string, it may be left out. That is, the use of OPN is optional. If two OPN's are used in a block operation specification the smaller of the two will control.

5. DATA MOVE OPERATIONS:

TRANS 0606 — Transfer-Nonblock

Syntax: 0606/FXOP/FXOP/

Copy a fixed point value from one location into another. The contents of the first location is not changed.

Special Case

Syntax: 0606/FLACC/FXOP/

0606/FXOP/FLACC/

Whenever the floating accumulator is specified as an argument of TRANS an unfloat or float operation is performed during the transfer.

XCHNG 0607 — Exchange - Nonblock

Syntax: 0607/FXOP/FXOP/

Exchange the contents of two fixed point data locations.

Special Case

Syntax: 0606/FLACC/FXOP/

0606/FXOP/FLACC/

Whenever the floating accumulator is specified as an argument of TRANS an unfloat or float operation is performed during the transfer.

BLKTR 0624 — Block Transfer

Syntax: 0624/BLOCK/INDEX/OPN/BLOCK/INDEX/OPN/

Special Case: 0624/BLOCK/INDEX/OPN/FLACC/

Special Case: 0624/FLACC/BLOCK/INDEX/OPN/

Copy a string of data from one block to another - the originating block is not changed.

Special Cases: If the first element of a block is specified to be transferred to the second element of the same block, the entire block will be set to the value of the first element. I. E., the transfer occurs in ascending iterations. Similarly, "Rippling" can be accomplished within a block of data.

RBLTR 0311 Reverse Block Transfer

Syntax: 0311/BLOCK/INDEX/OPN/BLOCK/INDEX/OPN/

This function is quite similar to block transfer (key 0624). The operation is different in that the transfer iteration takes place starting at the end of the data block and proceeds in a descending order. This operation is included to allow the "rippling" of data within a block in a different manner. It is suggested that this function not be used where the "Block Transfer" function (key 0624) could be used, because the time required is considerably greater. It is not nearly so great, however, as a loop of instructions to accomplish the same result.

Note that FLACC may not be used as an operand.)

XCHNG 0312 Exchange Block Transfer

Syntax: 0312/BLOCK/INDEX/OPN/BLOCK/INDEX/OPN/

The function of this operation is to exchange the contents of two strings of data. The operation is performed in ascending iterations from the beginnings of the specified blocks (as modified by index).

Note that special cases can result when a string is exchanged with itself starting at different locations.
Note the special case of FLACC as an operand produces a predictable but not usual result. (It is recommended that FLACC not be used as an operand.)

ZOPR 0412 ZERO-FILL Operand

Syntax: 0412/BLOCK/

Store a zero into every word of the specified operand.
Note that "Index" and "OPN" are not allowed.

6. MISCELLANEOUS OPERATIONS

BP1 0403 **Breakpoint 1**

Syntax: BP1/

If toggle 1 is up for the station executing this operation, then stop the computer. Start may be pressed to continue. If toggle 1 is down then execute the next sequential key operation.

BP2 0404 **Breakpoint 2**

Same as BP1 (0403) except uses toggle 2.

BP3 0405 **Breakpoint 3**

Same as BP1 (0403) except uses toggle 3.

BP4 0406 **Breakpoint 4**

Same as BP1 (0403) except uses toggle 4.

BP5 0407 **Breakpoint 5**

Same as BP1 (0403) except uses toggle 5.

BP6 0410 **Breakpoint 6**

Same as BP1 (0403) except uses toggle 6.

PAUSE 0425 **PAUSE**

Syntax: PAUSE

This operation, when executed, causes the computer to stop. It may be restarted by pressing the start switch.

7. ARITHMETIC AND LOGARITHMIC OPERATIONS

Fixed point arithmetic is integer type 14 bit plus sign (MODULO 32768). Arithmetic is performed with true sign control. When an overflow occurs (only ADD SUB, MUL, and DIVIDE BY ZERO can cause overflow) the fixed point overflow indicator is set. This indicator can be tested and reset with a jump operation.

Floating point arithmetic and functions are carried out by subroutines. A floating point number is carried as three words, the first being a power of two exponent, excess 1024, for the second and third words which are the most and least significant portions of a fraction. Detached sign convention is used and is carried in the sign position of the most significant fraction word. Conventional sign control is performed by each operation. Overflow conditions set the floating point overflow indicator. Underflow results in a zero answer.

Transfer of floating point data must be accomplished by a block-transfer operation.

ADDFL 0006 — Floating Add to FLACC

Syntax: 0006/FLOP/

Add the data from "FLOP" to "FLACC" and leave the result in "FLACC". Overflow turns on the floating overflow indicator.

SUBFL 0007 — Floating Subtract from FLACC

Syntax: 0007/FLOP/

Subtract the contents of "FLOP" from "FLACC" and leave the result in "FLACC". Overflow turns on the floating overflow indicator.

MULFL 0013 Floating Multiply with FLACC

Syntax: 0013/FLOP/

Multiply the contents of "FLOP" by "FLACC and leave the result in "FLACC". Overflow turns on the floating overflow indicator.

DIVFL 0014 Floating Divide into FLACC

Syntax: 0014/FLOP/

Divide "FLACC" by the contents of "FLOP". If divide by zero is attempted, the floating indicator is turned on.

SINE 0010 Sine of FLACC

Syntax: 0010/

The sine of "FLACC" (which must be in radians) is computed and stored in "FLACC". Overflow may occur.

COSINE 0015 Cosine of FLACC

Syntax: 0015/

The cosine of "FLACC" (which must be in radians) is computed and stored in "FLACC". Overflow may occur.

ARCTAN0011 Arctangent of FLACC

Syntax: 0011/

The arctangent of "FLACC" is computed and stored in "FLACC" (as radians).

LOGAR 0012 Logarithm, Base 10, of FLACC

Syntax: 0012/

Compute the base 10 logarithm of "FLACC" and store the result in "FLACC". If "FLACC" was zero or negative, set the floating overflow indicator on.

ANTLOG 0017 Base 10 Antilog of FLACC

Syntax: 0017/

Compute the base 10 antilog of "FLACC" and store the result in "FLACC". Overflow may occur.

SQROOT 0016 Square Root of FLACC

Syntax: 0016/

The square root of "FLACC" is stored in "FLACC". If "FLACC" was negative, the floating overflow indicator is set on.

CHSFL 0023 Change Sign of FLACC

Syntax: 0023/

Change the sign of FLACC.

ABSFL 0024 Absolute FLACC

Syntax: 0024/

Make the sign of FLACC positive.

HXFLA 0004 Hollerith to FLACC

Syntax: 0004/BLOCK/INDEX/OPN/

This operation converts a string of characters to floating point and leaves the result in "FLACC". The sign and decimal point may be placed anywhere in the word. The number string is terminated on the first non-sign, non-decimal point, non-numeric, character, or when "OPN" characters have been handled, or at the end of "BLOCK"; whichever comes first. No sign is interpreted as positive.

FLAH 0005 FLACC to Hollerith

Syntax: 0005/BLOCK/INDEX/OPN/

This operation converts "FLACC" to a string of characters and stores the string in a regular block data location, right justified. The number is terminated after seven digits. Leading spaces are supplied until the field is nine or less characters long (7 digits plus sign and decimal point). The block data location will be filled with X's if the digit string exceeds the specified field

length.

Floating Point No.	5 Char. Field	7 Char. Field	11 Char-Field
1.0	␣ 1.00	␣ 1.0000	␣␣␣ 1.000000
1234.	␣ XXX	␣ 1234.0	␣␣␣ 1234.000
-.1234	-.123	-.12340	␣␣ -.1234000
12.3456789	␣12.3	␣ 12.345	␣␣␣ 12.34567
.000012	␣ .000	␣ .00001	␣␣␣ .0000120
-12345.	-XXXX	-12345.	␣␣ -12345.00

␣ = SPACE

ADDFX 0106 — Fixed Add to FXACC

Syntax: 0106/FXOP

Add the contents of "FXOP" to "FXACC". If overflow results, turn on the fixed point overflow indicator.

SUBFX 0107 — Fixed Subtract from FXACC

Syntax: 0107/FXOP/

Subtract the contents of "FXOP" from "FXACC". If overflow results, turn on the fixed point overflow indicator.

MUL 0113 — Fixed Multiply by FXACC

Syntax: 0113/FXOP/

Multiply the contents of "FXACC" by "FXOP". If overflow results, turn on the fixed point overflow indicator.

DIV 0114 — Fixed Divide into FXACC

Syntax: 0114/FXOP/

Divide the contents of "FXACC" by "FXOP". The integer quotient, unrounded, is place in "FXACC".

If divide by zero is attempted, the fixed point overflow indicator is set on, and $7FFF_{16}$ (largest possible fixed point number) is placed in "FXACC".

INCR 0123 — Increment

Syntax: 0123/FXOP/

Add one to "FXOP". Overflow will not occur. Arithmetic is modulo 32,768.

DECR 0124 — Decrement

Syntax: 0124/FXOP/

Subtract one from "FXOP", overflow will not occur. Arithmetic is modulo 32,768.

XTRCT 0135 — Extract

Syntax: 0135/FXOP/

Perform a bit by bit logical "AND" with "FXOP" and "FXACC".

Leave result in "FXACC".

Example: FXACC = 0011

FXOP = 0110

FXACC = 0010

HFXA 0035 — Hollerith to Fixed Accumulator.

Syntax: 0035/BLOCK/INDEX/OPN/

The operand for this function specifies the beginning of a sequence of words containing decimal numbers. The operation is to binarize the string of numbers and place the result in the fixed point accumulator. The first valid character may be preceded by spaces which are ignored. A sign + or - may be anywhere in the string of numbers. If more than one sign

is present, the last one encountered will control the sign of the result. No sign is taken to mean positive. Binarization will continue until a non-numeric, nonsign character or until the end of the operand containing the string is encountered. If overflow results, the fixed point overflow indicator will be set.

FXAH 0036 Fixed Accumulator to Hollerith

Syntax: 0036/BLOCK/INDEX/OPN/

The operand for this function specifies the beginning of a sequence of words. The operation is debinarize (convert to a string of BCD characters) the contents of the fixed point accumulator and store the result in the specified string of words left justified. The sign of the accumulator is stored in the first word of the string (- for minus, space for plus). The most significant character of the number is stored in the next word, the next number is the next word, etc., until up to five characters have been stored. If the specified string has remaining words in it, they are filled with spaces. Whenever the end of the string is encountered, the operation is terminated regardless of whether the complete debinarized number string has been stored. No indication is made to indicate the fact, therefore, it is the responsibility of the console programmer to provide a large enough string for the number to be debinarized (six is always sufficient).

FX1CMP 0130 — Twos Complement Fixed Accumulator

Syntax: 0130/

Convert the contents of "FXACC" to twos complement form and store in "FXACC".

FX2CMP 0131 — Ones Complement Fixed Accumulator

Syntax: 0131/

Convert the contents of "FXACC" to ones complement form and store in "FXACC".

CMPGFX 0125 Compare Fixed Greater Than

Syntax: 0125/FXOP/

Compare "FXOP" and "FXACC".

If "FXACC" is greater than "FXOP" set comparator indicator true.

If "FXACC" is less than or equal to "FXOP" set comparator indicator false.

CMPEFX 0126 Compare Fixed Equal

Syntax: 0126/FXOP/

Compare "FXOP" and "FXACC".

If "FXACC" is equal to "FXOP" set comparator indicator true.

If "FXACC" is not equal to "FXOP" set comparator indicator false.

CMPLFX 0127 Compare Fixed Less Than

Syntax: 0127/FXOP/

If "FXACC" is less than "FXOP" set comparator indicator true.

If "FXACC" is greater than or equal to "FXOP" set comparator indicator false.

8. LOGICAL OPERATIONS

CFLG 0025 — Compare Floating Greater Than

Syntax: 0025/FLOP/

Compare "FLOP" and "FLACC"

If "FLACC" is greater than "FLOP" set comparator indicator true.

If "FLACC" is equal to "FLOP" do not change comparator indicator.

If "FLACC" is less than "FLOP" set comparator indicator false.

CFLL 0027 — Compare Floating Less Than

Syntax: 0027/FLOP/

Compare "FLOP" and "FLACC".

If "FLACC" is less than "FLOP" set comparator indicator true.

If "FLACC" is equal to "FLOP" do not change comparator indicator.

If "FLACC" is greater than "FLOP" set comparator indicator false.

SRCHNH 325 — Search Numeric High

Syntax: 0325/BLOCK/INDEX/OPN/

Search a specified block starting at the point specified by index. The first time the contents of the fixed point accumulator (FXACC) is numerically higher in value than the value of the contents of the block word being tested, the search operation is stopped. When this condition is satisfied FXACC is set to the number of tests made minus one.

If the condition is not met, FXACC is set to minus one.

SRCHEQ 326 — Search Equal

Syntax: 0326/BLOCK/INDEX/OPN/

Search a specified block starting at the point specified by index. The first time the contents of the fixed point accumulator (FXACC) is equal in value to the value of the contents of the block word being tested, the search operation is stopped. When this condition is satisfied FXACC is set to the number of tests made minus one. If the condition is not met, FXACC is set to minus one.

SRCHNL 327 — Search Numeric Low

Syntax: 0327/BLOCK/INDEX/OPN

Search a specified block starting at the point specified by index. The first time the contents of the fixed point accumulator (FXACC) is numerically lower in value than the value of the contents of the block word being tested, the search operation is stopped. When this condition is satisfied FXACC is set to the number of tests made minus one. If the condition is not met, FXACC is set to minus one.

SRCHNQ 330 — Search Not Equal

Syntax: 0330/BLOCK/INDEX/OPN

Search a specified block starting at the point specified by index. The first time the contents of the fixed point accumulator (FXACC) is not equal in value to the value of the contents of the block word being tested, the search operation is stopped. When this condition is satisfied FXACC is set to the number

of tests made minus one. If the condition is not met, FXACC is set to minus one.

SETRU 0030 — Set Comparator True

Syntax: 0030/

Set the comparator indicator true.

SETFL 0031 — Set Comparator False

Syntax: 0031

Set the comparator indicator false.

9. CONTROL OPERATIONS

The jump keys operate in conjunction with the "Label Follows" function (key 0521). If a jump is active (i.e., the jump criteria is true) then program control is transferred to the operation immediately following the "Label" specified by the jump syntax.

JFLPOS 0020 — Jump If "FLACC" is Positive

Syntax: 0020/LABEL/

JFLNEG 0022 — Jump If "FLACC" is Negative

Syntax: 0021/LABEL/

JFLOV 0032 — Jump If the Floating Overflow Indicator Is On

Syntax: 0032/LABEL/

JFXOV 0033 — Jump If the Fixed Overflow Indicator Is On

Syntax: 0033/LABEL/

JFXPOS 0120 — Jump If "FXACC" is Positive

Syntax: 0120/LABEL/

JFXZRO 0121 — Jump If "FXACC" is Zero

Syntax: 0121/LABEL/

JFXNEG 0122 — Jump If "FXACC" is Negative

Syntax: 0122/LABEL/

JTRU 0132 — Jump If Comparator Indicator is True

Syntax: 0132/LABEL/

JFLS 0133 — Jump If Comparator Indicator is False

Syntax: 0133/LABEL/

JALL 0134 — Jump Always

Syntax: 0134/LABEL/

10. INPUT/OUTPUT OPERATIONS

MT2K 0034 — Read Mag Tape Into 2K Buffer

Syntax: 0034/

Read a block of data from mag tape (in binary format) into the 2K buffer. A short block will result in the latter part of the A2KOP (2048-block length) not being changed. A long block will result in data being lost.

If an EOF is read, the fixed overflow indicator is set on and the A2KOP contents are undisturbed.

If an EOF is not read, the fixed overflow indicator is set off.

RDMT 0323 — Read Mag Tape Into An Operand

Syntax: 0323/BLOCK/INDEX/OPN/

This operation is similar to MT2K (key 0034) except the A2KOP is replaced by the specified block (which may be the A2KOP).

RDMTB 0603 — Read Mag Tape Into the Mag Tape Buffer

Syntax: 0603/

This operation is similar to MT2K (key 0034) except the A2KOP is replaced by the mag tape buffer (key 0617). A further feature is offered wherein alpha-mode recording is done if the sign bit of the MTR (key 0615) is a 0. if it is a 1 binary-mode recording is done.

RDEOF 0604 — Read Mag Tape To An End-Of-File

Syntax: 0604/

This operation causes the mag tape to move forward until an EOF (End-of-File) mark is sensed. The tape

is then in a position to read the record following the EOF. (The fixed overflow indicator is not changed).

A2KMT 0136 — Write Mag Tape From 2K Buffer

Syntax: 0136/

Write a 2048 word, binary mode, record from the 2K buffer onto mag tape.

WRMT 0324 — Write Mag Tape From An Operand

Syntax: 0324/BLOCK/INDEX/OPN/

Write a binary mode record from the specified operand. The record is as long as OPN or block length-index whichever is shorter.

WRMTB 0610 — Write Mag Tape From The Mag Tape Buffer

Syntax: 0610/

Write an 80 word record from the mag tape buffer. The sign bit of the MTR (key 0615) controls the mode of writing, 0 for Alpha-Mode or 1 for Binary Mode.

WREOF 0611 — Write A Mag Tape End-Of-File Mark

Syntax: 0611/

Write an end-of-file mark on mag tape.

RWNMT 0605 — Rewind Mag Tape

Syntax: 0605/

Rewind the specified mag tape.

BKSP 0612 — Backspace Mag Tape

Syntax: 0612/

Backspace the mag tape one record.

WRLP 0627

Write Line Printer

Syntax: 0627/BLOCK/INDEX/OPN/

This key, when executed, causes the contents of the specified "operand" to be output on the line printer. The line printer will accommodate 120 characters per line. When printing from the operand, the least significant 6 bits only, of each word, are translated to appropriate characters. When the word containing the first character to be output also contains a sign bit, the line printer will skip to the first line of the next page before printing. When the sign bit of the first word is not present, the line printer advances to the next line and prints. The next line after the last line of a page is considered to be the first line of the next page. Thus at page ends there is about a one-inch unprinted space.

11. DISPLAY OPERATIONS

WRDSP 0636 — Write Display

Syntax: 0636/

This key, when executed, causes the contents of the display module buffer to be transferred to the display console CRT. Because the display module memory is addressable, the system must supply beginning and ending addresses of where the data is to be stored. This is accomplished by placing the beginning address in the 79th position of the display buffer or DMFR (Key 0316) register and the ending address in the 78th position or DMTO (Key 0317) register. The ending address may be less than 78 greater than the beginning address, in which case only a portion of the display buffer will be output to the console. In the case of the ending address being exactly 78 greater than the beginning address, the entire buffer contents will be output. If the ending address is more than 78 greater than the beginning address, the contents of the display buffer will be completely output and the operation terminated; i.e., never will more than the complete buffer be output and no indication is given if more is called for.

RDDSP 0635 — Read Display

Syntax: 0635/

The key causes the display module's memory to be read into the display buffer. The beginning and ending address to be read from and to must be placed in the DMB elements 79 and 78 respectively, or optionally,

the addresses may be placed in the DMFR (Key 0316) and the DMTO (Key 0317) registers. (See description for keys 0316, 0317.)

READA 0430 — Read Page A to the A2KOP

Syntax: 0430/

The console display memory for Page A is copied into the "A2KOP" (Key 0423). If the BR-85 is the display console, reading begins at cell 16 (20 octal); if the BR-90 is the display console reading begins at cell 0 of the display memory.

WRTA 0431 — Write Page A From The A2KOP

Syntax: 0431

See Read A (Key 0430). Note it is considered an illegal operation to "WRTA" to the BR-90, if the BR-90 is in "Formatted" mode.

READB 0432 — Read Page B To The A2KOP

Syntax: 0432/

The console display memory for Page B is copied into the "A2KOP"(Key 0423).

WRTB 0433 — Write Page B From The A2KOP

Syntax: 0433/

The "A2KOP" (Key 0423) is copied into the console display memory for Page B. Note it is considered an illegal operation to "WRTB" if the BR-90 is in "formatted" mode.

RDOP 0207 — Read Display Memory to An Operand

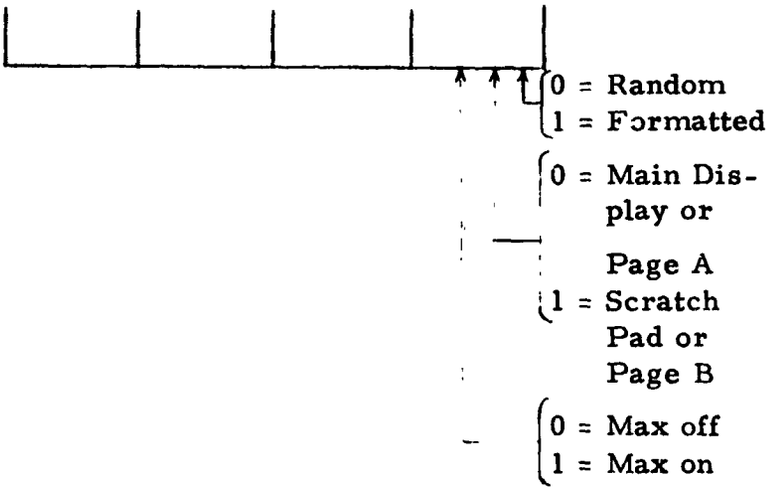
Syntax: 0207/BLOCK/INDEX/(PN/

The beginning and ending addresses of display memory must be set into DMFR (Key 0316) and DMTO (Key 0317) before execution of this operation. This

SM 0333 **Set Mode - BR-90 only**

Syntax: 0333/FXOP/FXOP/

The operation sets the BR-90 operating mode. The "FXOP" immediately follow 0333/ is coded as follows:



The second "FXOP" is a dummy and may contain any data as it is not used.

CPC 0332 **Control Projector - BR-90 Only**

Syntax: 0332/FXOP/FXOP/

This operation controls the projector mode. The "FXOP" immediately following 0332/ is coded as follows:

operation causes the specified display memory to be copied into the "BLOCK" indicated. The number of words copied is the smaller of: Block Dimension-Index, OPN, OR "DMTO" - "DMFR".

WROP 0210 — Write Display Memory From An Operand

Syntax: 0210/BLOCK/INDEX/OPN/

Similar to RDOP (Key 0207). Note it is considered an illegal operation to write into cells 0 and 1 when the BR-90 is in "Formatted" mode.

RPK 0623 — Read Program Key

Syntax: 0623/

This operation causes the computer to idle until a program key is entered. When the key is entered its signature is placed in the "SR" (Key 0003) and the idle mode is terminated.

RDCUR 0633 — Read Cursor

Syntax: 0633/

This operation causes the "CURX" (Key 0620) and "CURY" (Key 0621) registers to be loaded with the current cursor coordinates. The status of the display console is also placed in the "DMB" (Key 0616).

SETLI 0306 — Set Key Lights

Syntax: 0306/

This operation sets the program key lights as determined by the information in the "LTR" (Key 0305).

ALARM 0307 — Sound Alarm

Syntax: 0307/

This operation cuases the audible alarm of the console to sound for a short period of time.

NOTE

Bit	Content	
0-2	100 = Upper 010 = Center 001 = Lower	Mag Mode Quadrant Selection
3-5	100 = Left 010 = Center 001 = Right	
6	0 = Lamp Off 1 = Lamp On	
7	0 = Mag Mode 1 = Full Slide	

The second "FXOP" controls slide selection as follows:

Bit	Content
0-3	Units Slide Number Range 0-9
4-7	Tens Slide Number Range 0-9
8-10	Magazine Number Range 0-4

CLM 0320 Clear Display

Syntax: 0320/FXOP/FXOP/

This operation will cause the display to be cleared from the memory address in the first "FXOP". For BR-90, formatted mode only, clearing will stop at the memory address in the second "FXOP".

SC 335 Set Cursor

This key should be followed by two operand locations: These locations can be either a Basic System register (i.e. FXACC), any specific location in an operand (i.e. DMB OCT 15/) or a specified value (i.e. OCT 36/). The first operand is the X coordinate for the cursor, the second operand is the Y coordinate. These coordinates should be given in standard BR-90 display memory form - values from 0 to 7777. The last two bits of these coordinate values will take on the same meaning as any coordinate in memory. Therefore, the last two bits of these coordinate values should be zero. Otherwise, a blank cursor can be generated. Therefore, any value obtained from the console itself (such as from the status block) should be extracted with a value of 7774.

APPENDIX B

IDL Operating Procedures

1.1 GENERAL

The Interactive Display Language (IDL) concept was to be implemented on the Bunker Ramo 90 console (BR-90) which communicated with a CDC 1700 computer. Since this combination was altered during the course of the project it became necessary to use the card reader as the communications link to the CDC 1700. Therefore, where a key of the BR-90 console was used for an IDL input, a hollerith card with an IDL code is substituted. The card format and codes will be discussed below.

1.2 START UP PROCEDURE

It is assumed that the reader is familiar with the CDC 1700 system. To get IDL "on the air" the following procedures must be followed:

- o Load the IDL disk pack and depress the "START" switch. (Note: wait for the "O" switch light before continuing).

These next operations take place at the CDC 1700 console.

- o Hit the "CLEAR" switch up and down.
- o Depress the "AUTOLOAD" indicator.
- o Hit the "RUN" switch

The following sequence of operations will be performed on the tele-writer(TTY). The user response will be underlined.

```
PP
* CR          (depress "MANUAL INTERRUPT")
* MI
* P CR
* IDL CR
```

IDL is now ready to accept user programs through the card reader.

When a program is to be run the program cards should be in the card reader in the "READY" mode. All the user need now do is hit the "SELECTIVE SKIP" switch on the CDC 1700 console and the IDL program will be read in.

The "SELECTIVE SKIP" switch becomes the user "GO" response anytime "EXEC" gets control. (i.e., if an error occurred, the IDL program would be aborted and control returned to "EXEC". The user can now reload his program cards in the card reader and hit the "SELECTIVE SKIP" switch to start up again).

1.3 IDL INSTRUCTION KEYS

Figure B-1 contains all the available IDL instruction keys.

1.4 CARD FORMAT

In order for IDL programs to run the card format shown in Figure B-2 must be adhered to. The examples following should make clear the format specified. (See Figure B-3).

OVERLAY 000				
Key Type	OCT Key	HEX Key	Mnemonic	Syntax Type
1	1	1	OV+1	Special
1	2	2	OV-1	Special
4	3	3	SR	0
2	4	4	HFLA	3
2	5	5	FLAH	3
2	6	6	ADDFL	4
2	7	7	SUBFL	4
2	10	8	SINE	0
2	11	9	ARCTAN	0
2	12	A	LOGAR	0
2	13	B	MULFL	4
2	14	C	DIVFL	4
2	15	D	COSINE	0
2	16	E	SQROOT	0
2	17	F	ANTLOG	0
2	20	10	JFLPOS	0
1	21	11	UNDIG	0
2	22	12	JFLNEG	0
2	23	13	CHSFL	0
2	24	14	ABSFL	0
2	25	15	CFLG	0
2	26	16	CAE	5
2	27	17	CFLL	4
2	30	18	SETRU	0
2	31	19	SETFL	0
2	32	1A	JFLOV	Special
2	33	1B	JFXOV	Special
2	34	1C	A2KMT	0
2	35	1D	HFXA	3
2	36	1E	FXAH	3

Figure B-1

OVERLAY 001				
Key Type	OCT Key	HEX Key	Mnemonic	Syntax Type
1	1	21	OV+1	Special
1	2	22	OV-1	Special
4	3	23	X9REG	NA
4	4	24	X8REG	NA
4	5	25	X7REG	NA
2	6	26	ADDFX	4
2	7	27	SUBFX	4
4	10	28	X6REG	NA
4	11	29	X5REG	NA
4	12	2A	X4REG	NA
2	13	2B	MUL	4
2	14	2C	DIV	4
4	15	2D	X3REG	NA
4	16	2E	X2REG	NA
4	17	2F	X1REG	NA
3	20	30	JFXPOS	Special
3	21	31	JFX2RO	Special
3	22	32	JFXNEG	Special
2	23	33	INCR	4
2	24	34	DECR	4
2	25	35	CMPGFX	4
2	26	36	CMPEFX	4
2	27	37	CMPFLX	4
2	30	38	FX2CMP	0
2	31	39	FX1CMP	0
3	32	3A	JTRU	Special
3	33	3B	JFLS	Special
3	34	3C	JALL	Special
2	35	3D	XTRCT	4
2	36	3E	MT2K	0

Figure B-1 (Cont'd)

OVERLAY 002				
Key Type	OCT Key	HEX Key	Mnemonic	Syntax Type
1	1	41	OV+1	Special
1	2	42	OV-1	Special
0	3	43	Unused	
0	4	44	Unused	
0	5	45	Unused	
0	6	46	Unused	
2	7	47	RDOP	0
2	10	48	WROP	3
2	11	49	DATIN	3
2	12	4A	DATOO	3
0	13	4B	Unused	
0	14	4C	Unused	
0	15	4D	Unused	
0	16	4E	Unused	
0	17	4F	Unused	
0	20	50	Unused	
0	21	51	Unused	
0	22	52	Unused	
0	23	53	Unused	
0	24	54	Unused	
0	25	55	Unused	
0	26	56	Unused	
0	27	57	Unused	
0	30	58	Unused	
0	31	59	Unused	
0	32	5A	Unused	
2	33	5B	SHLL8	0
0	34	5C	Unused	
2	35	5D	STFX	4
2	36	5E	LDFX	4

Figure B-1 (Cont'd)

OVERLAY 003				
Key Type	OCT Key	HEX Key	Mnemonic	Syntax Type
1	1	61	OV+1	Special
1	2	62	OV-1	Special
0	3	63	Unused	
0	4	64	Unused	
8	5	65	LTR	NA
2	6	66	SETLI	0
2	7	67	ALARM	0
2	10	68	ERRCV	0
2	11	69	RBLTR	1
2	12	6A	XBLK	1
0	13	6B	Unused	
0	14	6C	Unused	
0	15	6D	Unused	
4	16	6E	DMFR	NA
4	17	6F	DMTO	NA
2	20	70	CLM	2
1	21	71	Unused	
0	22	72	Unused	
2	23	73	RDMT	3
2	24	74	WRMT	3
2	25	75	SRCHNH	3
2	26	76	SRCHEQ	3
2	27	77	SRCHNL	3
2	30	78	SRCHNQ	3
1	31	79	Unused	
2	32	7A	CPC	2
2	33	7B	SM	2
0	34	7C	Unused	
2	35	7D	SC	4
0	36	7E	Unused	

Figure B-1 (Cont'd)

OVERLAY 004				
Key Type	OCT Key	HEX Key	Mnemonic	Syntax Type
1	1	81	OV+1	Special
1	2	82	OV-1	Special
2	3	83	BP1	0
2	4	84	BP2	0
2	5	85	BP3	0
2	6	86	BP4	0
2	7	87	BP5	0
2	10	88	BP6	0
2	11	89	ABORT	0
2	12	8A	ZOPR	3
0	13	8B	Unused	
0	14	8C	Unused	
0	15	8D	Unused	
1	16	8E	LNPRT	NA
0	17	8F	Unused	
1	20	90	DISPL	0
2	21	91	CALL	5
0	22	92	Unused	
8	23	93	A2KOP	NA
0	24	94	Unused	
2	25	95	PAUSE	0
0	26	96	Unused	
1	27	97	ASY2K	0
2	30	98	READA	0
2	31	99	WRTA	0
2	32	9A	READB	0
2	33	9B	WRTB	0
0	34	9C	Unused	
0	35	9D	Unused	
0	36	9E	Unused	

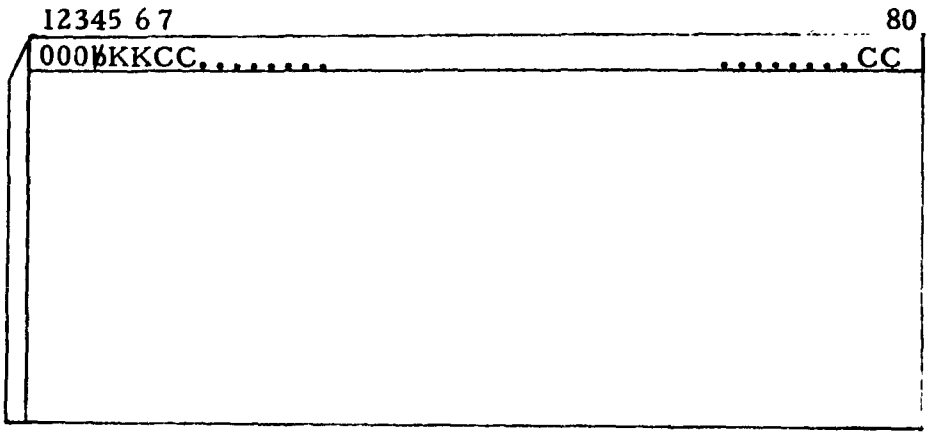
Figure B-1 (Cont'd)

OVERLAY 005				
Key Type	OCT Key	HEX Key	Mnemonic	Syntax Type
1	1	A1	OV+1	Special
1	2	A2	OV-1	Special
F	3	A3	DECFL	Special
F	4	A4	DEC	Special
F	5	A5	OCT	Special
1	6	A6	START	Special
1	7	A7	STOP	Special
4	10	A8	SREG	NA
F	11	A9	DECPT	NA
F	12	AA	NEG	NA
1	13	AB	DEFIN	0
2	14	AC	END	0
F	15	AD	9	NA
F	16	AE	8	NA
F	17	AF	7	NA
2	20	B0	SHLR8	0
3	21	B1	LF	0
F	22	B2	6	NA
F	23	B3	5	NA
F	24	B4	4	NA
1	25	B5	ASIGN	0
1	26	B6	UNDEF	0
1	27	B7	3	NA
F	30	B8	2	NA
F	31	B9	1	NA
1	32	BA	EXECU	Special
1	33	BB	SUBST	Special
F	34	BC	INFIN	1
F	35	BD	0	NA
F	36	BE	/	Spe

Figure B-1 (Cont'd)

OVERLAY 006				
Key Type	OCT Key	HEX Key	Mnemonic	Syntax Type
1	1	C1	OV+1	Special
1	2	C2	OV-1	Special
2	3	C3	RDMTB	0
2	4	C4	RDEOF	0
2	5	C5	RWNMT	0
2	6	C6	TRANS	2
2	7	C7	XCHNG	2
2	10	C8	WRMTB	0
2	11	C9	WREOF	0
2	12	CA	BKSP	0
4	13	CB	FLACC	NA
4	14	CC	FXACC	NA
0	15	CD	Unused	
8	16	CE	DMB	NA
8	17	CF	MTB	NA
4	20	D0	CURX	NA
4	21	D1	CURY	NA
4	22	D2	LGR	NA
2	23	D3	RPK	0
2	24	D4	BLKTR	1
0	25	D5	Unused	
0	26	D6	Unused	
2	27	D7	WRLP	3
0	30	D8	Unused	
0	31	D9	Unused	
2	32	DA	RDCD	3
2	33	DB	RDCUR	0
0	34	DC	Unused	0
2	35	DD	RDDSP	0
2	36	DE	WRDSP	0

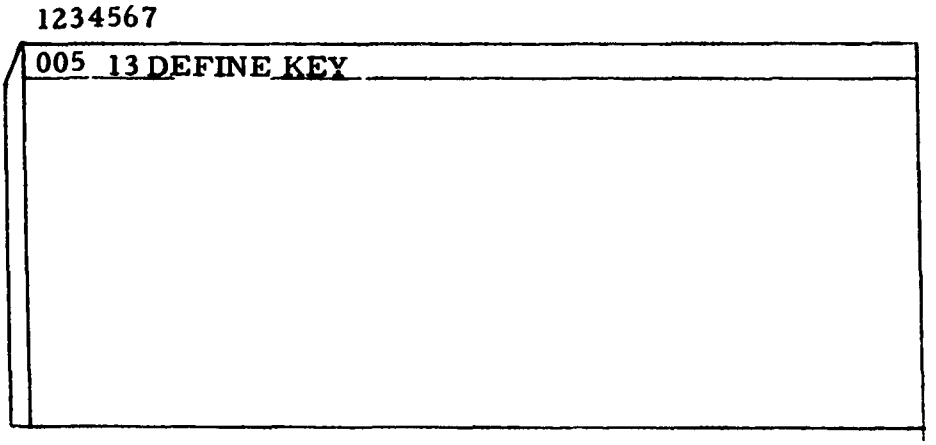
Figure B-1 (Cont'd)



COLUMNS	CONTENT
1-3	Octal Overlay number (0-6)
4	Space (blank)
5-6	Octal Key number (1-36)
7-80	Comments

Card Format
Figure B-2

EXAMPLE 1



EXAMPLE 2

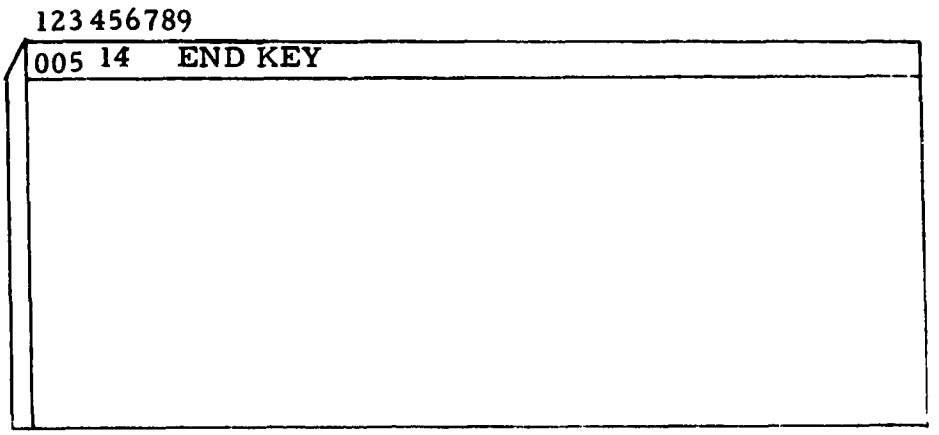


Figure B-3

1.5 SAMPLE PROGRAMS

PROGRAM 1

```
004 16   SET LINE PRINTER FLAG  **LIST PROGRAM**
005 13   DEFINE KEY    **DEFINE PROGRAM NAME**
002 14   UNASSIGNED KEY
005 25   ASSIGN KEY    **DEFINE STORAGE**
002 17   UNASSIGNED KEY
005 31   NUMBER 1
005 36   DELIMITER
005 14   END KEY
005 25   ASSIGN KEY
002 20   UNASSIGNED KEY
005 30   NUMBER 2
005 31   NUMBER 1
005 36   DELIMITER
005 04   DECIMAL CONSTANT
005 22   NUMBER 6
005 36   DELIMITER
005 04   DECIMAL CONSTANT
005 23   NUMBER 5
005 36   DELIMITER
005 04   DECIMAL CONSTANT
005 24   NUMBER 4
005 36   DELIMITER
005 04   DECIMAL CONSTANT
005 27   NUMBER 3
005 36   DELIMITER
005 04   DECIMAL CONSTANT
005 30   NUMBER 2
```

```
005  36.  DELIMITER
005  14   END KEY
005  25   ASSIGN KEY
002  15   UNASSIGNED KEY
005  31   NUMBER 1
005  36   DELIMITER
005  14   END KEY
005  25   ASSIGN KEY
002  16   UNASSIGNED KEY
005  31   NUMBER 1
005  36   DELIMITER
005  14   END KEY
002  36   LOAD FXACC  **START PROGRAM**
002  20   BLOCK 20
005  05   OCTAL CONSTANT
005  30   NUMBER 2  (INDEX)
005  36   DELIMITER
001  06   ADDFX   (ADD FIXED POINT)
005  04   DECIMAL CONSTANT
005  27   NUMBER 3
005  36   DELIMITER
001  07   SUBFX   (SUBTRACT FXD PNT)
002  20   BLOCK 20
005  04   DECIMAL CONSTANT
005  31   NUMBER 1  (INDEX)
005  36   DELIMITER
002  35   STORE FXACC
002  17   LOCATION 17
002  33   SHIFT LEFT 8
```

```
001  14   DIVIDE FXACC
002  20   BLOCK 20
005  05   OCTAL CONSTANT
005  24   NUMBER 4  (INDEX)
005  36   DELIMITER
002  35   STORE FXACC
002  16   LOCATION 16
005  14   END PROGRAM
```

PROGRAM 2

```
004  16   SET LINE PRINTER FLAG   **LIST PROGRAM**
005  13   DEFINE KEY   **DEFINE PROGRAM NAME**
002  14   UNASSIGNED KEY
005  25   ASSIGN KEY   **DEFINE STORAGE**
002  20   UNASSIGNED KEY
005  30   NUMBER 2
005  31   NUMBER 1
005  36   DELIMITER
005  04   DECIMAL CONSTANT
005  22   NUMBER 6
005  36   DELIMITER
005  05   OCTAL CONSTANT
005  23   NUMBER 5
005  36   DELIMITER
005  04   DECIMAL CONSTANT
005  24   NUMBER 4
005  36   DELIMITER
005  04   DECIMAL CONSTANT
005  27   NUMBER 3
```

```
005 36   DELIMITER
005 04   DECIMAL CONSTANT
005 30   NUMBER 2
005 36   DELIMITER
005 14   END KEY
005 25   ASSIGN KEY
002 15   UNASSIGNED KEY
005 31   NUMBER 1
005 36   DELIMITER
005 14   END KEY
005 25   ASSIGN KEY
002 16   UNASSIGNED KEY
005 31   NUMBER 1
005 36   DELIMITER
005 14   END KEY
005 25   ASSIGN KEY
002 17   UNASSIGNED KEY
005 31   NUMBER 1
005 36   DELIMITER
005 14   END KEY
005 25   ASSIGN KEY
002 21   BLOCK 21
005 16   NUMBER 8
005 36   DELIMITER
005 05   OCTAL CONSTANT
005 24   NUMBER 4
005 35   NUMBER 0
005 23   NUMBER 5
005 24   NUMBER 4
```

```
005  35   NUMBER 0
005  36   DELIMITER
005  05   OCTAL CONSTANT
005  35   NUM 0
005  36   DELIMITER
005  05   OCTAL CONSTANT
005  24   NUMBER 4
005  35   NUMBER 0
005  17   NUMBER 7
005  35   NUMBER 0
005  35   NUMBER 0
005  36   DELIMITER
005  05   OCTAL CONSTANT
005  35   NUMBER 0
005  36   DELIMITER
005  14   END KEY
006  24   BLOCK TRANSFER   **START PROGRAM**
002  20   BLOCK 20
005  05   OCTAL CONSTANT
005  35   NUMBER 0
005  36   DELIMITER
005  04   DECIMAL CONSTANT
005  31   NUMBER 1
005  15   NUMBER 9
005  36   DELIMITER
006  17   MAGTAPE BUFFER
005  05   OCTAL CONSTANT
005  35   NUMBER 0
005  36   DELIMITER
005  04   DECIMAL CONSTANT
```

```
005 31   NUMBER 1
005 16   NUMBER 8
005 36   DELIMITER
006 10   WRITE MAGTAPE BUFFER ON MAGTAPE
006 11   WRITE EOF MARK
006 05   REWIND MAGTAPE
004 12   ZERO FILL
006 17   MAGTAPE BUFFER
005 04   DECIMAL CONSTANT
005 35   NUMBER 0
005 36   DELIMITER
005 04   DECIMAL CONSTANT
005 31   NUMBER 1
005 16   NUMBER 8
005 36   DELIMITER
006 03   READ MAGTAPE INTO MAGTAPE BUFFER
006 12   BACKSPACE MAGTAPE
005 14   END PROGRAM
```

APPENDIX C

IDL Errors

1.1 EXECUTIVE ERRORS

ERROR TYPE	DESCRIPTION
2	Input Key too large - must be $1\text{-}36_8$
3	Input Overlay too large - must be 1-176
4	Key to be executed is not a program key

1.2 ASSEMBLER ERRORS

ERROR TYPE	DESCRIPTION
A1	Unidentified mode value (not a programmer error)
A2	An assigned key was used for program ID
A3	Tried to assign an assigned key
A4	Tried to preset more cells than assigned
A5	END Key encountered when a constant delimeter was still needed
A10	More than one parameter for JUMP command
A11	Too many parameters for this syntax type
A12	Same as A11
A13	Illegal key encountered while processing a constant
A14	Tried to use an unassigned key as parameter
A15	Illegal key encountered while processing a constant
A16	Syntax type not 0-6 (not a programmer error)

ERROR TYPE	DESCRIPTION
A17	Illegal syntax for syntax Type 1 command
A18	Illegal syntax for syntax Type 2 command
A19	Illegal syntax for syntax Type 3 command
A20	Illegal syntax for syntax Type 4 command
A21	Illegal syntax for syntax Type 5 command
A22	Illegal syntax for syntax Type 6 command
A23	Destination of JUMP command not declared as a label
A24	End of subroutine not found
A30	Constant processing flag set but constant type flag not set (not a programmer error)
A31	Constant not begun with assemble constant key
A32	Delimeter not received before expected
A33	Two negation keys received
A34	Two infinity keys received
A35	Illegal key - constant already infinity
A36	Illegal key - too many digits
A37	Illegal key - not a constant type key (not a programmer error)
A38	Constant too large - overflow
A39	Same as A38
A40	Illegal constant - no digits
A41	Same as A40
A42	Two decimal points received
A43	Same as A43
A44	Too many digits in constant
A45	Same as A37

ERROR TYPE	DESCRIPTION
A46	Floating point number too large - over-flow

1.3 LOADER ERRORS

ERROR TYPE	DESCRIPTION
L1	Storage packet identifier was not in the proper place
L2	Constant Packet Identifier was not in proper place
L3	Jump Packet Identifier not in proper place
L4	Command Packet not first word
L5	No label found in Jmp packet for label in command string
L6	Disk error unable to load physical malfunction

1.4 INTERPRETER ERRORS

ERROR TYPE	DESCRIPTION
I1	Bad sequence of command
I2	Bad JUMP-Command key type 3 but unknown test
I3	Bad JUMP - Undefined Label
I4	Bad parameter in command packet

NOTE

Error messages are outputted on the TTY in the form

ERROR TYPE XXX

(XXX is the error type number listed in Appendix C.) When an error does occur, it refers to the last printed command on the line printer. All commands before the erroneous one are error-free.